1. Introduction

Inventory levels can make or break a business. they able to Get a smart view of your stock with inventory planning, optimization and management software. Effective stock control is essential for any business to operate efficiently. Keeping track of inventory, managing orders, and ensuring accurate stock levels are crucial for seamless operations and customer satisfaction. With advancements in technology, many companies are turning to software development services to streamline their stock control processes and improve efficiency. Accurate stock control is vital for businesses to meet customer demands and avoid costly errors. Inaccurate inventory levels can lead to stockouts, overstocking, delayed deliveries, and unhappy customers. By implementing software solutions tailored to your specific stock control needs, you can significantly reduce errors and improve overall efficiency. Custom software development services can provide you with a personalized system that automates tasks, tracks inventory levels in real-time, and generates accurate reports for informed decision-making.

1.1 Project Scope

To undertake this project, we conducted a meeting with a tire company. During the visit, we inquired about the general operational processes of the tire shop, including how they classify tires, when they place new orders, the various types of tires they handle, and the criteria customers use when selecting tires. We proposed developing a program based on the information gathered, and the tire shop owner agreed.

We developed a scoring system according to the information obtained from the tire shop. This prototype aims to create an inventory control system tailored to their needs. The system is designed to respond to user requirements based on a scoring system that evaluates tires according to the criteria specified by the company. The most critical criteria for the company's inventory control include the sales rates of tires, expiration dates, and the number of tires in stock.

In accordance with the company's requirements, we developed a system that allows for the storage, analysis, and visualization of various brands and models of tires within the warehouse. Additionally, based on the information received from the tire company, our system can also indicate products that require discounts within the warehouse. This system helps in efficiently managing tire inventory, ensuring that the company can meet customer demands and optimize stock levels.

1.2 Importance of Tire Management Systems

A tire management system is crucial for efficiently handling tire inventory, tracking usage, and optimizing reorder processes. It helps in maintaining optimal stock levels, reducing waste, and ensuring that the right tires are available when needed.

2. System Design and Architecture

In this section, we will discuss the design and architecture of the Tire Management System. This includes the structural features, components, and interactions of the system, explaining their basic functionalities.

2.1.Overall System Architecture

The Tire Management System consists of the following main components:

1. Main Class: The entry point of the system.

The main class is the central control unit of the Tire Management System application. It is responsible for initializing the system components, managing user interactions, handling various operations such as adding, removing, updating, and displaying tires, and providing an interface for the user.

```java
import java.time.format.DateTimeParseException;
import java.util.ArrayList;
import java.util.List;

public class Main {
    private static final ArrayList<Tire> tires = new ArrayList<>();
    private static Storage storage;
    private static final JFrame frame = new JFrame("Tire Inventory Management");

    public static void main(String[] args) {
        // Warehouse list
        Warehouse newCarWarehouse = new Warehouse("New Car Warehouse");
        Warehouse oldCarWarehouse = new Warehouse("Old Car Warehouse");
        Warehouse newExcavationWarehouse = new Warehouse("New Excavation Warehouse");
        Warehouse oldExcavationWarehouse = new Warehouse("Old Excavation Warehouse");
        Warehouse newAgricultureWarehouse = new Warehouse("New Agriculture Warehouse");
        Warehouse oldAgricultureWarehouse = new Warehouse("Old Agriculture Warehouse");
        storage = new Storage(newCarWarehouse, oldCarWarehouse, newExcavationWarehouse, oldExca

        addInitialData();
        createMainMenu();
    }
```

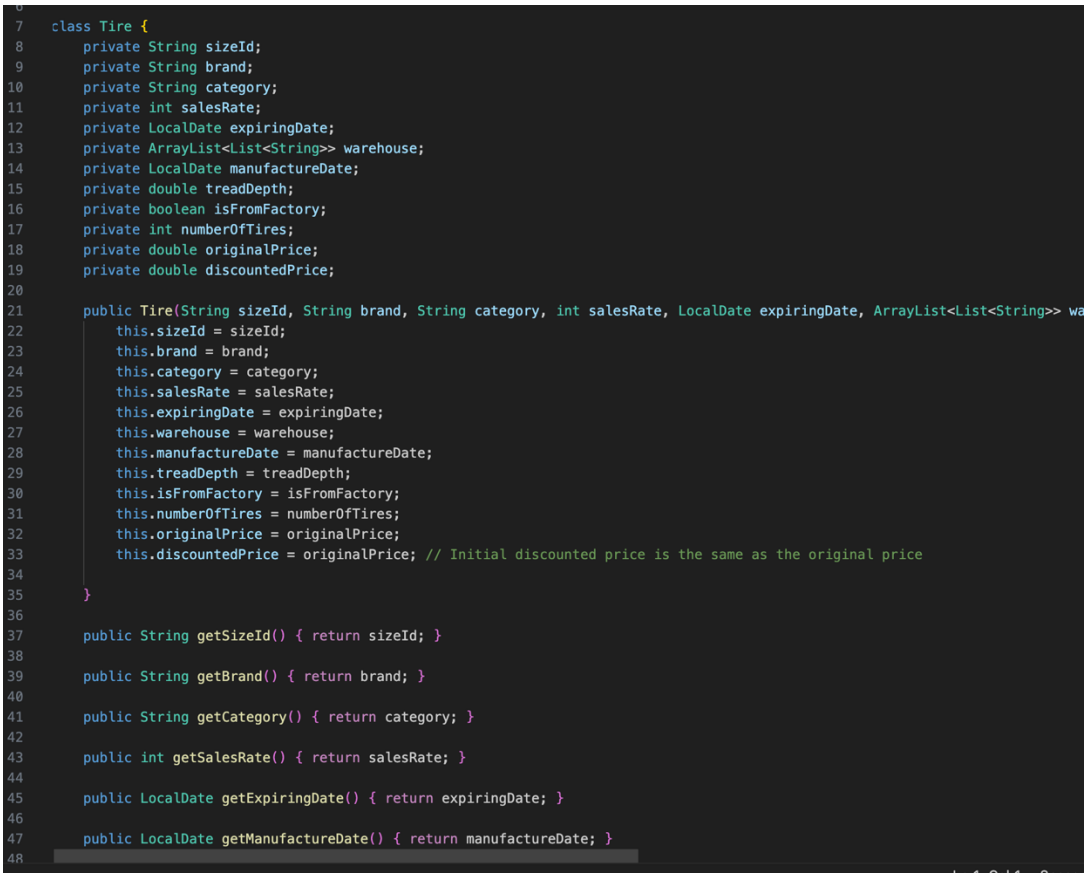
2. Storage Class: Manages the tire inventory.

The Storage class is responsible for managing the storage and categorization of tires. It interacts with multiple Warehouse instances, each corresponding to a specific category and condition of tires.The `Storage` class initializes six warehouses to store new and old tires across three categories: car, excavation, and agriculture. It determines whether a tire is new or old based on its manufacture date and tread depth, and then stores the tire in the appropriate warehouse according to its category and condition.

```
7   public class Storage {
25      public void storeTire(Tire tire) {
26          long daysSinceManufacture = ChronoUnit.DAYS.between(tire.getManufactureDate(), LocalDate.now());
27          boolean isNew = tire.isFromFactory() || (tire.getTreadDepth() >= 7 && daysSinceManufacture <= 30);
28
29          if (tire.getCategory().equals("car")) {
30              if (isNew) {
31                  carNewWarehouse.addTire(tire);
32              } else {
33                  carOldWarehouse.addTire(tire);
34              }
35          } else if (tire.getCategory().equals("excavation")) {
36              if (isNew) {
37                  excavationNewWarehouse.addTire(tire);
38              } else {
39                  excavationOldWarehouse.addTire(tire);
40              }
41          } else if (tire.getCategory().equals("agriculture")) {
42              if (isNew) {
43                  agriculturalNewWarehouse.addTire(tire);
44              } else {
45                  agriculturalOldWarehouse.addTire(tire);
46              }
47          }
48      }
49
50      public Warehouse getCarNewWarehouse() {
51          return carNewWarehouse;
52      }
53
54      public Warehouse getCarOldWarehouse() {
55          return carOldWarehouse;
56      }
57
58      public Warehouse getExcavationNewWarehouse() {
59          return excavationNewWarehouse;
60      }
61
62      public Warehouse getExcavationOldWarehouse() {
63          return excavationOldWarehouse;
64      }
65
66      public Warehouse getAgriculturalNewWarehouse() {
```

3. Tire Class: Represents the tire entity.

The Tire class is central to the Tire Management System, encapsulating all the necessary attributes and behaviors related to a tire. It manages all attributes related to a tire, such as size, brand, category, sales rate, expiration date, warehouse location, manufacture date, tread depth, factory status, number of tires, original price, and discounted price. By encapsulating tire-related data, the Tire class ensures data integrity and security. Additionally, it manages the original and discounted prices of tires, providing comprehensive price management capabilities.



```
class Tire {
    private String sizeId;
    private String brand;
    private String category;
    private int salesRate;
    private LocalDate expiringDate;
    private ArrayList<List<String>> warehouse;
    private LocalDate manufactureDate;
    private double treadDepth;
    private boolean isFromFactory;
    private int numberOfTires;
    private double originalPrice;
    private double discountedPrice;

    public Tire(String sizeId, String brand, String category, int salesRate, LocalDate expiringDate, ArrayList<List<String>> wa
        this.sizeId = sizeId;
        this.brand = brand;
        this.category = category;
        this.salesRate = salesRate;
        this.expiringDate = expiringDate;
        this.warehouse = warehouse;
        this.manufactureDate = manufactureDate;
        this.treadDepth = treadDepth;
        this.isFromFactory = isFromFactory;
        this.numberOfTires = numberOfTires;
        this.originalPrice = originalPrice;
        this.discountedPrice = originalPrice; // Initial discounted price is the same as the original price

    }

    public String getSizeId() { return sizeId; }

    public String getBrand() { return brand; }

    public String getCategory() { return category; }

    public int getSalesRate() { return salesRate; }

    public LocalDate getExpiringDate() { return expiringDate; }

    public LocalDate getManufactureDate() { return manufactureDate; }
```

4. TireDSS Class: Provides decision support functionalities.

The Tire Management System is designed to manage the storage and categorization of various types of tires. This section provides an overview of the system's architecture, focusing on the TireDSS class and its role within the system. The TireDSS class is a critical component of the Tire Management System, providing decision support functionalities that help in managing tire inventories effectively. The responsibilities of the TireDSS class include displaying tires, suggesting orders, and providing analysis. It offers functionality to display the list of tires stored in various warehouses, analyzes tire inventory levels to suggest orders and maintain optimal stock, and provides insights and analysis on tire sales, inventory levels, and other relevant metrics.

```java
public DetailedScore calculateScore(Tire tire) {
    int totalScore = 0;
    int salesRate = tire.getSalesRate();
    int numberOfTires = tire.getNumberOfTires();
    long daysToExpiry = ChronoUnit.DAYS.between(LocalDate.now(), tire.getExpiringDate());

    boolean highSalesRate = salesRate >= 60;
    boolean lowTireCount = numberOfTires < 12;
    boolean expirySoon = daysToExpiry < 30;

    int salesRateContribution = 0;
    int salesRateAndLowTireCountContribution = 0;
    int salesRateLowTireCountExpiryContribution = 0;
    int lowTireCountContribution = 0;
    int lowTireCountAndExpiryContribution = 0;
    int expirySoonContribution = 0;

    // Scoring system
    if (highSalesRate && lowTireCount) {
        salesRateAndLowTireCountContribution = 5 * salesRate;
        totalScore += salesRateAndLowTireCountContribution;
    }
    if (highSalesRate && lowTireCount && expirySoon) {
        salesRateLowTireCountExpiryContribution = 4 * salesRate;
        totalScore += salesRateLowTireCountExpiryContribution;
    }
    if (lowTireCount && expirySoon) {
        lowTireCountAndExpiryContribution = 3 * salesRate;
        totalScore += lowTireCountAndExpiryContribution;
    }
    if (lowTireCount) {
        lowTireCountContribution = 2 * salesRate;
        totalScore += lowTireCountContribution;
    }
    if (expirySoon) {
        expirySoonContribution = 1 * salesRate;
        totalScore += expirySoonContribution;
    }
```

5. Warehouse Class: Mediates between storage and decision support.

The Warehouse class is a critical component of the Tire Management System, responsible for organizing and storing tires based on their categories and conditions. The responsibilities of the Warehouse class include adding tires to the inventory, removing tires from the inventory, and retrieving tire data. It offers functionality to categorize tires, maintain accurate records of tire inventories, and ensure that tires are stored under optimal conditions. The Tire Management System's architecture is designed to efficiently manage tire inventories through the use of the Warehouse class. The Warehouse class plays a crucial role in adding and removing tires from the inventory and providing access to tire data, ensuring that the system operates effectively and efficiently. This modular design ensures that the system is robust, maintainable, and scalable.

```java
package org.example;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Warehouse {

    private final String name;

    private final Map<String, List<Tire>> tires;

    public Warehouse(String name) {
        this.name = name;
        this.tires = new HashMap<>();
    }

    public void addTire(Tire tire) {
        tires.computeIfAbsent(tire.getBrand(), k -> new ArrayList<>()).add(tire);
    }

    public Map<String, List<Tire>> getAllTires() {
        return tires;
    }

    public Map<String, List<Tire>> getTires() {
        return tires;
    }

    public String getName() {
        return name;
    }
}

```

3 Detailed Analysis of the Stock Management Code

3.1 Storage Class

The Storage class is designed to handle the storage and retrieval of tire data from a data source, such as a database or file system. This class ensures that the tire data is persistently stored and can be efficiently retrieved for further processing and analysis.

Here is the complete definition of the Storage class based on the provided file

```java
public class Storage {

    private final Warehouse carNewWarehouse;
    private final Warehouse carOldWarehouse;
    private final Warehouse excavationNewWarehouse;
    private final Warehouse excavationOldWarehouse;
    private final Warehouse agriculturalNewWarehouse;
    private final Warehouse agriculturalOldWarehouse;

    public Storage(Warehouse carNewWarehouse, Warehouse carOldWarehouse, Warehouse excavationNewWarehouse, Warehouse excavationOldWare
        this.carNewWarehouse = carNewWarehouse;
        this.carOldWarehouse = carOldWarehouse;
        this.excavationNewWarehouse = excavationNewWarehouse;
        this.excavationOldWarehouse = excavationOldWarehouse;
        this.agriculturalNewWarehouse = agriculturalNewWarehouse;
        this.agriculturalOldWarehouse = agriculturalOldWarehouse;
    }

    public void storeTire(Tire tire) {
        long daysSinceManufacture = ChronoUnit.DAYS.between(tire.getManufactureDate(), LocalDate.now());
        boolean isNew = tire.isFromFactory() || (tire.getTreadDepth() >= 7 && daysSinceManufacture <= 30);

        if (tire.getCategory().equals("car")) {
            if (isNew) {
                carNewWarehouse.addTire(tire);
            } else {
                carOldWarehouse.addTire(tire);
            }
        } else if (tire.getCategory().equals("excavation")) {
            if (isNew) {
                excavationNewWarehouse.addTire(tire);
            } else {
                excavationOldWarehouse.addTire(tire);
            }
        } else if (tire.getCategory().equals("agriculture")) {
            if (isNew) {
                agriculturalNewWarehouse.addTire(tire);
            } else {
                agriculturalOldWarehouse.addTire(tire);
            }
        }
    }
```

```
    public Warehouse getCarNewWarehouse() {
        return carNewWarehouse;
    }

    public Warehouse getCarOldWarehouse() {
        return carOldWarehouse;
    }

    public Warehouse getExcavationNewWarehouse() {
        return excavationNewWarehouse;
    }

    public Warehouse getExcavationOldWarehouse() {
        return excavationOldWarehouse;
    }

    public Warehouse getAgriculturalNewWarehouse() {
        return agriculturalNewWarehouse;
    }

    public Warehouse getAgriculturalOldWarehouse() {
        return agriculturalOldWarehouse;
    }
}
```

The Storage class manages the placement of tires in six different warehouses based on category and newness. Tires are classified as new or old according to production date and tread depth and added to appropriate warehouses. This class ensures proper placement of tires in warehouses and access to these tanks when necessary. This system simplifies warehouse management by ensuring tires are classified and stored correctly.

3.2.Detailed Analysis of the Stock Management

Class	**Description**	**Attributes/Methods**
TireDSS	Main class responsible for calculating the score of a tire based on certain criteria.	calculateScore(Tire tire) - Method that computes the score for a given tire.
DetailedScore	Class that encapsulates the detailed breakdown of the score calculated for a tire.	Attributes totalScore, salesRateContribution, salesRateAndLowTireCountContribution, salesRateLowTireCountExpiryContribution, lowTireCountContribution, lowTireCountAndExpiryContribution, expirySoonContribution. Methods: Getters for each attribute.

```java
public class TireDSS {

    public DetailedScore calculateScore(Tire tire) {
        int totalScore = 0;
        int salesRate = tire.getSalesRate();
        int numberOfTires = tire.getNumberOfTires();
        long daysToExpiry = ChronoUnit.DAYS.between(LocalDate.now(), tire.getExpiringDate());

        boolean highSalesRate = salesRate >= 60;
        boolean lowTireCount = numberOfTires < 12;
        boolean expirySoon = daysToExpiry < 30;

        int salesRateContribution = 0;
        int salesRateAndLowTireCountContribution = 0;
        int salesRateLowTireCountExpiryContribution = 0;
        int lowTireCountContribution = 0;
        int lowTireCountAndExpiryContribution = 0;
        int expirySoonContribution = 0;

```

Line	Code Fragment	Description	Relationships
1	public DetailedScore calculateScore(Tire tire)	Method definition for calculating the detailed score of a tire.	Takes a Tire object as input and returns a DetailedScore object.
2	int totalScore = 0;	Initializes the total score to 0.	totalScore is the cumulative score for the tire, which will be updated based on conditions.
3	int salesRate = tire.getSalesRate();	Retrieves the sales rate of the tire.	Calls getSalesRate method on tire object.
4	int numberOfTires = tire.getNumberOfTires();	Retrieves the number of tires in stock.	Calls getNumberOfTires method on tire object.
5	long daysToExpiry = ChronoUnit.DAYS.between(Local	Calculates the number	Uses ChronoUnit.DAYS.between to calculate days between

	Date.now(), tire.getExpiringDate());	of days until the tire expires by finding the difference between the current date and the expiration date.	current date and expiration date.
6	boolean highSalesRate = salesRate >= 60;	Checks if the sales rate is high (i.e., greater than or equal to 60).	Compares salesRate value to 60.
7	boolean lowTireCount = numberOfTires < 12;	Checks if the tire count is low (i.e., less than 12).	Compares numberOfTires value to 12.
8	boolean expirySoon = daysToExpiry < 30;	Checks if the tire is expiring soon (i.e., within 30 days).	Compares daysToExpiry value to 30.
9	int salesRateContribution = 0;	Initializes the contribution of sales rate	salesRateContribution will be updated based on conditions involving salesRate.

		to the total score to 0.	
10	int salesRateAndLowTireCountContribution = 0;	Initializes the contribution when both sales rate is high and tire count is low to 0.	salesRateAndLowTireCountContribution will be updated based on conditions involving both salesRate and lowTireCount.
11	int salesRateLowTireCountExpiryContribution = 0;	Initializes the contribution when sales rate is high, tire count is low, and tire is expiring soon to 0	salesRateLowTireCountExpiryContribution will be updated based on conditions involving salesRate, lowTireCount, and expirySoon.
12	int lowTireCountContribution = 0;	Initializes the contribution of low tire count to the total score to 0.	lowTireCountContribution will be updated based on conditions involving lowTireCount.
13	int lowTireCountAndExpiryContribution = 0;	Initializes the contribution when both tire count is low and tire is expiring	lowTireCountAndExpiryContribution will be updated based on conditions involving both lowTireCount and expirySoon.

		soon to 0.	
14	int expirySoonContribution = 0;	Initializes the contribution of expiring soon factor to the total score to 0	expirySoonContribution will be updated based on conditions involving expirySoon.

The `calculateScore` method is designed to compute a detailed score for a tire, providing a quantitative assessment based on various factors. This method starts by initializing the total score to zero. It retrieves essential attributes of the tire, such as the sales rate using `tire.getSalesRate()`, the number of tires in stock using `tire.getNumberOfTires()`, and calculates the days until the tire expires by computing the difference between the current date and the expiration date using `ChronoUnit.DAYS.between(LocalDate.now(), tire.getExpiringDate())`.

The method then evaluates several boolean conditions: whether the sales rate is high (`salesRate >= 60`), whether the tire count is low (`numberOfTires < 12`), and whether the tire is expiring soon (`daysToExpiry < 30`). These conditions are used to determine the contributions to the total score.

Several contribution variables are initialized to zero: `salesRateContribution`, `salesRateAndLowTireCountContribution`, `salesRateLowTireCountExpiryContribution`, `lowTireCountContribution`, and `lowTireCountAndExpiryContribution`. Each of these contributions is updated based on specific conditions:

- `salesRateContribution` is updated if the sales rate is high.

- `salesRateAndLowTireCountContribution` is updated if both the sales rate is high and the tire count is low.

- `salesRateLowTireCountExpiryContribution` is updated if the sales rate is high, the tire count is low, and the tire is expiring soon.

- `lowTireCountContribution` is updated if the tire count is low.

- `lowTireCountAndExpiryContribution` is updated if both the tire count is low and the tire is expiring soon.

These contributions are then summed up to calculate the total score, which is encapsulated in a `DetailedScore` object and returned by the method. This scoring system allows for a

nuanced evaluation of each tire, taking into account multiple factors that affect its urgency and demand, thereby aiding in prioritizing inventory management decisions.

```
// Scoring system
if (highSalesRate && lowTireCount) {
    salesRateAndLowTireCountContribution = 5 * salesRate;
    totalScore += salesRateAndLowTireCountContribution;
}
if (highSalesRate && lowTireCount && expirySoon) {
    salesRateLowTireCountExpiryContribution = 4 * salesRate;
    totalScore += salesRateLowTireCountExpiryContribution;
}
if (lowTireCount && expirySoon) {
    lowTireCountAndExpiryContribution = 3 * salesRate;
    totalScore += lowTireCountAndExpiryContribution;
}
if (lowTireCount) {
    lowTireCountContribution = 2 * salesRate;
    totalScore += lowTireCountContribution;
}
if (expirySoon) {
    expirySoonContribution = 1 * salesRate;
    totalScore += expirySoonContribution;
}
```

The scoring system consists of several conditional checks that update various contribution variables and the totalScorebased on specific criteria. Here's a detailed breakdown of each condition and its impact on the score.

The scoring system shown in the image assigns scores to tires based on a combination of their sales rate, stock levels, and expiration dates. The system uses a series of conditional checks to determine the contribution of each factor to the total score. If a tire has both a high sales rate (greater than or equal to 60) and a low stock level (less than 12), the `salesRateAndLowTireCountContribution` is calculated as five times the sales rate and added to the total score. If, in addition to these conditions, the tire is expiring soon (within 30 days), the `salesRateLowTireCountExpiryContribution` is calculated as four times the sales rate and added to the total score.

For tires that have a low stock level and are expiring soon, the `lowTireCountAndExpiryContribution` is calculated as three times the sales rate and added to the total score. If a tire has only a low stock level, the `lowTireCountContribution` is calculated as two times the sales rate and added to the total score. Finally, if a tire is only expiring soon, the `expirySoonContribution` is calculated as the sales rate and added to the total score.

This hierarchical scoring system ensures that tires with a combination of high sales, low stock, and imminent expiration receive the highest priority, guiding inventory management decisions to focus on the most urgent and high-demand items. By using multipliers based on the sales

rate, the system dynamically adjusts the contributions, reflecting the relative importance of each factor in the context of tire management.

3.3Stock management Conditions

Condition	Value	Requirement Situation	Actions
boolean highSalesRate= salesRate	>= 60	if (highSalesRate &&lowTireCount) if (highSalesRate && lowTireCount && expirySoon)	salesRateAndLowTireCountContribution =5 * salesRate;totalScore+= salesRateAndLowTireCountContribution; salesRateLowTireCountExpiryContribution=4 * salesRate;totalScore+= salesRateLowTireCountExpiryContribution;
boolean lowTireCount = numberOfTires	< 12	if (lowTireCount && expirySoon) if (lowTireCount)	lowTireCountAndExpiryContribution = 3 * salesRate; totalScore += lowTireCountAndExpiryContribution; lowTireCountContribution = 2 * salesRate; totalScore += lowTireCountContribution;
boolean expirySoon= daysToExpiry	< 30		expirySoonContribution = 1 * salesRate; totalScore += expirySoonContribution;

```java
class DetailedScore {
    private final int totalScore;
    private final int salesRateContribution;
    private final int salesRateAndLowTireCountContribution;
    private final int salesRateLowTireCountExpiryContribution;
    private final int lowTireCountContribution;
    private final int lowTireCountAndExpiryContribution;
    private final int expirySoonContribution;

    public DetailedScore(
            int totalScore,
            int salesRateContribution,
            int salesRateAndLowTireCountContribution,
            int salesRateLowTireCountExpiryContribution,
            int lowTireCountContribution,
            int lowTireCountAndExpiryContribution,
            int expirySoonContribution) {
        this.totalScore = totalScore;
        this.salesRateContribution = salesRateContribution;
        this.salesRateAndLowTireCountContribution = salesRateAndLowTireCountContribution;
        this.salesRateLowTireCountExpiryContribution = salesRateLowTireCountExpiryContribution;
        this.lowTireCountContribution = lowTireCountContribution;
        this.lowTireCountAndExpiryContribution = lowTireCountAndExpiryContribution;
        this.expirySoonContribution = expirySoonContribution;
    }
```

```java
    public int getTotalScore() {
        return totalScore;
    }

    public int getSalesRateContribution() {
        return salesRateContribution;
    }

    public int getSalesRateAndLowTireCountContribution() {
        return salesRateAndLowTireCountContribution;
    }

    public int getSalesRateLowTireCountExpiryContribution() {
        return salesRateLowTireCountExpiryContribution;
    }

    public int getLowTireCountContribution() {
        return lowTireCountContribution;
    }

    public int getLowTireCountAndExpiryContribution() {
        return lowTireCountAndExpiryContribution;
    }

    public int getExpirySoonContribution() {
        return expirySoonContribution;
    }
}
```

Purpose: The DetailedScore class stores the width details of the tires.

Constructor: This constructor takes the units of all units and assigns them to its variables in the class.

3.4DetailedScore Class Analysis

Class	Attributes	Requiriment Functions
DetailedScore	int totalScore,	public int getTotalScore() { return totalScore; }
DetailedScore	int salesRateContribution,	public int getSalesRateContribution() { return salesRateContribution; }
DetailedScore	int salesRateAndLowTireCountContribution,	public int getSalesRateAndLowTireCountContribution() { return salesRateAndLowTireCountContribution; }
DetailedScore	İnt salesRateLowTireCountExpiryContribution,	public int getSalesRateLowTireCountExpiryContribution() { return salesRateLowTireCountExpiryContribution; }
DetailedScore	int lowTireCountContribution,	public int getLowTireCountContribution() { return lowTireCountContribution; }
DetailedScore	int lowTireCountAndExpiryContribution	public int getLowTireCountAndExpiryContribution() { return lowTireCountAndExpiryContribution; }
DetailedScore	int expirySoonContribution	public int getExpirySoonContribution() { return expirySoonContribution; }

The `DetailedScore` class is designed to encapsulate the various contributions to the total score of a tire, reflecting different aspects of its urgency and demand. This class has several attributes, each representing a specific type of contribution. The `totalScore` attribute holds the overall score, and it can be accessed using the `getTotalScore()` method. The `salesRateContribution`

attribute captures the contribution based on the tire's sales rate, retrievable through `getSalesRateContribution()`.

The `salesRateAndLowTireCountContribution` attribute combines the sales rate and the condition of having a low tire count, and its value can be accessed via `getSalesRateAndLowTireCountContribution()`. Similarly, the `salesRateLowTireCountExpiryContribution` attribute considers the sales rate, low tire count, and imminent expiry, with its corresponding getter method being `getSalesRateLowTireCountExpiryContribution()`.

For contributions based solely on the low tire count, the `lowTireCountContribution` attribute is used, accessible through `getLowTireCountContribution()`. When both low tire count and imminent expiry are factors, the `lowTireCountAndExpiryContribution` attribute comes into play, retrievable via `getLowTireCountAndExpiryContribution()`. Lastly, the `expirySoonContribution` attribute addresses the contribution due to the tire expiring soon, accessible through `getExpirySoonContribution()`. Each attribute and its corresponding getter method ensure that the detailed breakdown of the total score is available, facilitating a nuanced evaluation of each tire's status in the inventory.

3.5 Purpose and Impact of the Warehouse Class on Other Classes

The Warehouse class represents the main storage where tires are kept and managed. The purpose of this class is to organize, add, update, and manage stock information of tires in the warehouse. It interacts with other classes, playing a crucial role in the evaluation and scoring processes of the tires.

```java
package org.example;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Warehouse {

    private final String name;

    private final Map<String, List<Tire>> tires;

    public Warehouse(String name) {
        this.name = name;
        this.tires = new HashMap<>();
    }

    public void addTire(Tire tire) {
        tires.computeIfAbsent(tire.getBrand(), k -> new ArrayList<>()).add(tire);
    }

    public Map<String, List<Tire>> getAllTires() {
        return tires;
    }

    public Map<String, List<Tire>> getTires() {
        return tires;
    }

    public String getName() {
        return name;
    }
}
```

3.6 Warehouse Table

Component	Code	Description
Package Declaration	package org.example;	Declares the package name.
Imports	importjava.util.ArrayList; import java.util.HashMap; import java.util.List; import java.util.Map;	Imports the necessary Java utilities for lists and maps.
Class Declaration	public class Warehouse	Declares the Warehouse class.
Fields	private final String name; private final Map<String, List<Tire>> tires;	Declares two fields: name (name of the warehouse) and tires (a map of tire brands to lists of tires).

Constructor	public Warehouse(String name) {
	 this.name = name;
	 this.tires = new HashMap<>();
	 }
		Initializes the warehouse with a

		name and an empty map of tires.

addTire Method	public void addTire(Tire tire) {
	 tires.computeIfAbsent(tire.getBrand(), k -> new ArrayList<>()).add(tire);
	 }
		Adds a tire to the map, organizing by tire brand. If the brand does not exist, it creates a new list for that brand.

getAllTires Method	public Map<String, List<Tire>> getAllTires() {
	 return tires;
	 }
		Returns the map of all tires in the warehouse.

getTires Method	public Map<String, List<Tire>> getTires() {
	 return tires;
	 }
		Also returns the map of all tires in the warehouse (similar to getAllTires).

getName Method	public String getName() {
	 return name;
	 }
		Returns the name of the warehouse.

3.7 Conculusion Warehouse

The `Warehouse` class represents a scenario where tire inventory is managed by organizing tires based on their brand. When a `Warehouse` object is initialized with a name through its constructor, it sets the warehouse name and creates an empty `HashMap` to store tires. The `addTire` method allows adding a new tire to the warehouse by checking if the tire's brand already exists in the map. If not, it creates a new `ArrayList` for that brand and then adds the tire to the list. This method ensures that tires are organized by brand within the warehouse. The `getAllTires` and `getTires` methods provide access to the entire collection of tires stored in the warehouse, returning the map of tires organized by brand. Additionally, the `getName` method returns the warehouse's name, enabling other parts of the program to identify the warehouse. For example, creating a "Central Warehouse" and adding Michelin and Bridgestone tires to it allows the warehouse to manage its inventory efficiently. The tires are stored in lists categorized by their brands, and the warehouse name can be retrieved when needed. This class structure effectively facilitates tire inventory management, ensuring organized storage and easy retrieval of tire data.

4 Code Analysis

4.1 Storage.java

The Storage class manages the storage of tires by categorizing them into different warehouses based on their type and condition (new or old). It uses date and tread depth to determine if a tire is new and stores it accordingly.

```java
package org.example;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;


public class Storage {

    private final Warehouse carNewWarehouse;
    private final Warehouse carOldWarehouse;
    private final Warehouse excavationNewWarehouse;
    private final Warehouse excavationOldWarehouse;
    private final Warehouse agriculturalNewWarehouse;
    private final Warehouse agriculturalOldWarehouse;

    public Storage(Warehouse carNewWarehouse, Warehouse carOldWarehouse, Warehouse excavationNewWarehouse, Warehouse excavationOldWare
        this.carNewWarehouse = carNewWarehouse;
        this.carOldWarehouse = carOldWarehouse;
        this.excavationNewWarehouse = excavationNewWarehouse;
        this.excavationOldWarehouse = excavationOldWarehouse;
        this.agriculturalNewWarehouse = agriculturalNewWarehouse;
        this.agriculturalOldWarehouse = agriculturalOldWarehouse;
    }
```

```java
    public void storeTire(Tire tire) {
        long daysSinceManufacture = ChronoUnit.DAYS.between(tire.getManufactureDate(), LocalDate.now());
        boolean isNew = tire.isFromFactory() || (tire.getTreadDepth() >= 7 && daysSinceManufacture <= 30);

        if (tire.getCategory().equals("car")) {
            if (isNew) {
                carNewWarehouse.addTire(tire);
            } else {
                carOldWarehouse.addTire(tire);
            }
        } else if (tire.getCategory().equals("excavation")) {
            if (isNew) {
                excavationNewWarehouse.addTire(tire);
            } else {
                excavationOldWarehouse.addTire(tire);
            }
        } else if (tire.getCategory().equals("agriculture")) {
            if (isNew) {
                agriculturalNewWarehouse.addTire(tire);
            } else {
                agriculturalOldWarehouse.addTire(tire);
            }
        }
```

The `Storage` class manages multiple warehouses, each categorized by the type of tire and whether the tires are new or old. The class starts by importing necessary date and time utilities from `java.time`. It declares six private final `Warehouse` fields, each representing different warehouse categories: `carNewWarehouse`, `carOldWarehouse`, `excavationNewWarehouse`, `excavationOldWarehouse`, `agriculturaNewWarehouse`, and `agriculturaOldWarehouse`. The constructor initializes these fields with the respective `Warehouse` objects passed as parameters.

The `storeTire` method determines the appropriate warehouse for storing a given tire based on its category and condition. It calculates the days since the tire's manufacture using `ChronoUnit.DAYS.between` and checks if the tire is new, either by its factory status or tread depth and age. Depending on the tire's category (`car`, `excavation`, or `agriculture`) and whether it is new or old, the method adds the tire to the corresponding warehouse using the `addTire` method. This organized approach ensures that tires are stored systematically based on their type and condition, facilitating efficient inventory management.

4.2 Storage Management Table

Attribute/Method	Type	Description
carNewWarehouse	Warehouse	Warehouse for new car tires
carOldWarehouse	Warehouse	Warehouse for old car tires
xcavationNewWarehouse	Warehouse	Warehouse for new excavation tires
excavationOldWarehouse	Warehouse	Warehouse for old excavation tires
agriculturalNewWarehouse	Warehouse	Warehouse for new agricultural tires
agriculturalOldWarehouse	Warehouse	Warehouse for old agricultural tires
Constructor		**Initializes the warehouse fields**
Storage(...)	Constructor	Initializes carNewWarehouse, carOldWarehouse, excavationNewWarehouse, excavationOldWarehouse, agriculturalNewWarehouse, agriculturalOldWarehouse with provided values
Methods		**Description**
storeTire(Tire tire)	void	Stores a tire in the appropriate warehouse based on its category and condition
getCarNewWarehouse()	Warehouse	Returns the carNewWarehouse instance
getCarOldWarehouse()	Warehouse	Returns the carOldWarehouse instance
getExcavationNewWarehouse()	Warehouse	Returns the excavationNewWarehouse instance
getExcavationOldWarehouse()	Warehouse	Returns the excavationOldWarehouse instance
getAgriculturalNewWarehouse()	Warehouse	Returns the agriculturalNewWarehouse instance
getAgriculturalOldWarehouse()	Warehouse	Returns the agriculturalOldWarehouse instance

```
    public Warehouse getCarNewWarehouse() {
        return carNewWarehouse;
    }

    public Warehouse getCarOldWarehouse() {
        return carOldWarehouse;
    }

    public Warehouse getExcavationNewWarehouse() {
        return excavationNewWarehouse;
    }

    public Warehouse getExcavationOldWarehouse() {
        return excavationOldWarehouse;
    }

    public Warehouse getAgriculturalNewWarehouse() {
        return agriculturalNewWarehouse;
    }

    public Warehouse getAgriculturalOldWarehouse() {
        return agriculturalOldWarehouse;
    }
}
```

Description	Situation
Calculate days since manufacture	longdaysSinceManufacture= ChronoUnit.DAYS.between(tire.getManufactureDate(), LocalDate.now());
Determine if tire is new	`boolean isNew = tire.isFromFactory()
Check tire category	if (tire.getCategory().equals("car"))
Store car tire	if (isNew) { carNewWarehouse.addTire(tire); } else { carOldWarehouse.addTire(tire); }
Check excavation tire category	else if (tire.getCategory().equals("excavation"))
Store excavation tire	if (isNew) { excavationNewWarehouse.addTire(tire); } else { excavationOldWarehouse.addTire(tire); }
Check agricultural tire category	else if (tire.getCategory().equals("agriculture"))
Store agricultural tire	if (isNew) { agriculturalNewWarehouse.addTire(tire); } else { agriculturalOldWarehouse.addTire(tire); }

4.3 Tire.java

The Tire class is a representation of a tire with various attributes such as size, brand, category, sales rate, expiration date, warehouse details, manufacture date, tread depth, factory origin, number of tires, original price, and discounted price. The class provides a constructor to initialize these attributes and getter and setter methods to access and modify them.

```java
package org.example;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

class Tire {
    private String sizeId;
    private String brand;
    private String category;
    private int salesRate;
    private LocalDate expiringDate;
    private ArrayList<List<String>> warehouse;
    private LocalDate manufactureDate;
    private double treadDepth;
    private boolean isFromFactory;
    private int numberOfTires;
    private double originalPrice;
    private double discountedPrice;
```

4.4 Tire Attribute Table

Attribute	Type	Description
sizeId	String	Size identifier of the tire
brand	String	Brand of the tire
category	String	Category of the tire (e.g., car, excavation, agriculture)
salesRate	int	Sales rate of the tire
expiringDate	LocalDate	Expiration date of the tire
warehouse	ArrayList<List<String>>	Warehouse-related information stored as a list of lists
manufactureDate	LocalDate	Manufacture date of the tire
treadDepth	double	Depth of the tire tread
isFromFactory	boolean	Indicates if the tire is directly from the factory
numberOfTires	int	Number of tires
originalPrice	double	Original price of the tire
discountedPrice	double	Discounted price of the tire, initially set to the same as the original price

```java
public Tire(String sizeId, String brand, String category, int salesRate, LocalDate expiringDate, ArrayList<List<String>> warehouse
    this.sizeId = sizeId;
    this.brand = brand;
    this.category = category;
    this.salesRate = salesRate;
    this.expiringDate = expiringDate;
    this.warehouse = warehouse;
    this.manufactureDate = manufactureDate;
    this.treadDepth = treadDepth;
    this.isFromFactory = isFromFactory;
    this.numberOfTires = numberOfTires;
    this.originalPrice = originalPrice;
    this.discountedPrice = originalPrice; // Initial discounted price is the same as the original price
```

Constructor	Description
Tire(...)	When creating an object of the Tire class, this constructor method initializes all required fields of this object and makes the discountedPrice field the same as the originalPrice value. This ensures that all fields of the object have appropriate initial values.

```
    public String getSizeId() { return sizeId; }

    public String getBrand() { return brand; }

    public String getCategory() { return category; }

    public int getSalesRate() { return salesRate; }

    public LocalDate getExpiringDate() { return expiringDate; }

    public LocalDate getManufactureDate() { return manufactureDate; }

    public double getTreadDepth() { return treadDepth; }

    public boolean isFromFactory() { return isFromFactory; }

    public int getNumberOfTires() { return numberOfTires; }

    public double getOriginalPrice() { return originalPrice; }

    public double getDiscountedPrice() { return discountedPrice; }

    public void setDiscountedPrice(double discountedPrice) { this.discountedPrice = discountedPrice; }

    public void setNumberOfTires(int numberOfTires) {
        this.numberOfTires = numberOfTires;
    }

    public void setFromFactory(boolean fromFactory) {
        isFromFactory = fromFactory;
    }


    public void setOriginalPrice(double originalPrice) {
        this.originalPrice = originalPrice;
    }

}
```

The provided code snippet represents a set of getter and setter methods for a `Tire` class, which is designed to manage the attributes of a tire. The purpose of these methods is to encapsulate the tire's properties, allowing controlled access and modification. The getter methods retrieve various attributes such as `sizeId`, `brand`, `category`, `salesRate`, `expiringDate`, `manufactureDate`, `treadDepth`, `isFromFactory`, `numberOfTires`, `originalPrice`, and `discountedPrice`. Each getter method returns the value of the corresponding private field, ensuring that the data can be accessed but not modified directly.

On the other hand, the setter methods allow specific attributes to be modified. These include `setDiscountedPrice` for updating the discounted price of a tire, `setNumberOfTires` for changing the number of tires, `setFromFactory` for indicating if the tire is from the factory, and `setOriginalPrice` for setting the original price of the tire. These methods ensure that any changes to the tire's attributes are done through controlled and validated processes.

Overall, the purpose of this code is to provide a robust and secure way to manage the attributes of a tire, ensuring data integrity and encapsulation while allowing necessary modifications through well-defined methods. This structure supports the broader functionality of the tire inventory management system, facilitating efficient handling of tire data and operations.

4.5 Tire Method Table

Method	Return Type	Description
getSizeId()	String	Returns the size identifier of the tire
getBrand()	String	Returns the brand of the tire
getCategory()	String	Returns the category of the tire
getSalesRate()	int	Returns the sales rate of the tire
getExpiringDate()	LocalDate	Returns the expiration date of the tire
getManufactureDate()	LocalDate	Returns the manufacture date of the tire
getTreadDepth()	double	Returns the tread depth of the tire
isFromFactory()	boolean	Returns whether the tire is directly from the factory
getNumberOfTires()	int	Returns the number of tires
getOriginalPrice()	double	Returns the original price of the tire
getDiscountedPrice()	double	Returns the discounted price of the tire

-

Method	Parameters	Description
setDiscountedPrice(double price)	double price	Sets the discounted price of the tire
setNumberOfTires(int number)	int number	Sets the number of tires
setFromFactory(boolean factory)	boolean factory	Sets whether the tire is directly from the factory
setOriginalPrice(double price)	double price	Sets the original price of the tire

4.6 Main.java

Main.java

src > main > java > org > example > Main.java > ...

```
1   package org.example;
2
3   import javax.swing.*;
4   import java.awt.*;
5   import java.awt.event.ActionEvent;
6   import java.awt.event.ActionListener;
7   import java.time.LocalDate;
8   import java.time.format.DateTimeFormatter;
9   import java.time.format.DateTimeParseException;
10  import java.util.ArrayList;
11  import java.util.List;
12
13  public class Main {
14      private static final ArrayList<Tire> tires = new ArrayList<>();
15      private static Storage storage;
16      private static final JFrame frame = new JFrame(title:"Tire Inventory Management");
17
        Run | Debug
18      public static void main(String[] args) {…
31
32      private static void createMainMenu() {…
82
83      private static void addTire() {…
212
213
214
215     private static LocalDate readDate(String dateString) {…
224
225     private static void rankAndDisplayTires() {…
263
264
265     private static void checkOrderRequirements() {…
277
278     private static void applyAndDisplayDiscounts() {…
289
290     private static void displayWarehouses() {…
335
336     private static void addInitialData() {…
362 }
363
```

4.7 General Main Table

Component/Item	Description
Imports	Libraries and classes are imported. For example, javax.swing.*, java.awt.*, java.time.*, java.util.*, etc.
Class Definition (Main)	The main class is defined with public class Main {
ArrayList (tires)	private static final ArrayList<Tire> tires = new ArrayList<>(); An ArrayList is defined to store Tire objects.
Storage Object	private static Storage storage; The Storage object is defined for storage operations.
JFrame (frame)	private static final JFrame frame = new JFrame("Tire Inventory Management"); The JFrame object is defined for the GUI window and is given a title.
main() Method	public static void main(String[] args) {...} The main method, the entry point of the program.
createMainMenu() Method	private static void createMainMenu() {...} The method that creates the main menu panel.
addTire() Method	private static void addTire() {...} The method used to add a new tire.
readDate() Method	private static LocalDate readDate(String dateString) {...} The method used to read and convert date strings.
rankAndDisplayTires() Method	private static void rankAndDisplayTires() {...} The method that ranks and displays tires based on specified criteria
checkOrderRequirements() Method	private static void checkOrderRequirements() {...} The method that checks and displays order requirements.
applyAndDisplayDiscounts() Method	private static void applyAndDisplayDiscounts() {...} The method that applies discounts to selected tires and displays the updated information.
displayWarehouses() Method	private static void displayWarehouses() {...} The method that displays warehouses and their contents.
addInitialData() Method	private static void addInitialData() {...} The method that adds initial data, especially loading sample data when the program starts.

```
public static void main(String[] args) {
    // Warehouse list
    Warehouse newCarWarehouse = new Warehouse("New Car Warehouse");
    Warehouse oldCarWarehouse = new Warehouse("Old Car Warehouse");
    Warehouse newExcavationWarehouse = new Warehouse("New Excavation Warehouse");
    Warehouse oldExcavationWarehouse = new Warehouse("Old Excavation Warehouse");
    Warehouse newAgricultureWarehouse = new Warehouse("New Agriculture Warehouse");
    Warehouse oldAgricultureWarehouse = new Warehouse("Old Agriculture Warehouse");
    storage = new Storage(newCarWarehouse, oldCarWarehouse, newExcavationWarehouse, oldExcavationWarehouse, newAgricultureWarehous

    addInitialData();
    createMainMenu();
}
```

The `main` method in the provided code snippet serves as the entry point for the tire inventory management application. This method performs several crucial initialization tasks to set up the

application's environment. First, it creates instances of the `Warehouse` class for different categories and conditions of tires, including new and old car warehouses, new and old excavation warehouses, and new and old agriculture warehouses. Each of these `Warehouse` instances is initialized with a descriptive name that reflects its purpose.

After creating the warehouse instances, the `main` method initializes a `Storage` object by passing all the warehouse instances to its constructor. The `Storage` class is responsible for managing these warehouses collectively, facilitating operations across the different categories and conditions of tires.

The method then calls the `addInitialData()` function, which is likely responsible for populating the warehouses with initial tire data. This step is crucial for setting up a baseline inventory that the application can manage and manipulate.

Finally, the `createMainMenu()` function is called to set up the graphical user interface (GUI) for the application. This function likely creates and displays the main menu, allowing users to interact with the application through various options and controls.

In summary, the `main` method initializes the warehouse objects, sets up the storage system, populates initial data, and creates the main menu, thereby preparing the application for use.

Component	**Description**
Method Declaration	public static void main(String[] args) - The main method is the entry point of the program.
newCarWarehouse	Warehouse newCarWarehouse = new Warehouse("New Car Warehouse"); - Creates a new warehouse for new cars.
oldCarWarehouse	Warehouse oldCarWarehouse = new Warehouse("Old Car Warehouse"); - Creates a new warehouse for old cars.
newExcavationWarehouse	Warehouse newExcavationWarehouse = new Warehouse("New Excavation Warehouse"); - Creates a new warehouse for new excavation equipment.
oldExcavationWarehouse	Warehouse oldExcavationWarehouse = new Warehouse("Old Excavation Warehouse"); - Creates a new warehouse for old excavation equipment.
newAgricultureWarehouse	Warehouse newAgricultureWarehouse = new Warehouse("New Agriculture Warehouse"); - Creates a new warehouse for new agriculture equipment
oldAgricultureWarehouse	Warehouse oldAgricultureWarehouse = new Warehouse("Old Agriculture Warehouse"); - Creates a new warehouse for old agriculture equipment.

Storage Initialization	storage = new Storage(newCarWarehouse, oldCarWarehouse, newExcavationWarehouse, oldExcavationWarehouse, newAgricultureWarehouse, oldAgricultureWarehouse); - Initializes the storage object with the created warehouses.
addInitialData() Call	addInitialData(); - Calls the method to add initial data to the warehouses.
createMainMenu() Call	createMainMenu(); - Calls the method to create the main menu.

```
32      private static void createMainMenu() {
34          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35          frame.setSize(width:600, height:400);
36
37          JButton addButton = new JButton(text:"Add New Tire");
38          JButton displayButton = new JButton(text:"View Warehouses");
39          JButton rankButton = new JButton(text:"Sort and Show Tires");
40          JButton orderCheckButton = new JButton(text:"Check Order Requirements");
41          JButton discountButton = new JButton(text:"Apply and Show Discount");
42
43          addButton.addActionListener(new ActionListener() {
44              public void actionPerformed(ActionEvent e) {
45                  addTire();
46              }
47
48          JButton displayButton - org.example.Main.createMainMenu()
49          displayButton.addActionListener(new ActionListener() {
50              public void actionPerformed(ActionEvent e) {
51                  displayWarehouses();
52              }
53          });
54
55          rankButton.addActionListener(new ActionListener() {
56              public void actionPerformed(ActionEvent e) {
57                  rankAndDisplayTires();
58              }
59          });
60
61          orderCheckButton.addActionListener(new ActionListener() {
62              public void actionPerformed(ActionEvent e) {
63                  checkOrderRequirements();
64              }
65          });
66
67          discountButton.addActionListener(new ActionListener() {
68              public void actionPerformed(ActionEvent e) {
69                  applyAndDisplayDiscounts();
70              }
71          });
72
73          panel.add(addButton);
74          panel.add(displayButton);
75          panel.add(rankButton);
76          panel.add(orderCheckButton);
77          panel.add(discountButton);
```

The createMainMenu method sets up the main GUI by initializing and configuring a JFrame, adding five JButtons for various actions (add tire, view warehouses, sort and show tires, check order requirements, and apply and show discounts), attaching action listeners to these buttons to handle user interactions, and then adding the buttons to a JPanel.

```
83      private static void addTire() {
86
87          JPanel panel = new JPanel();
88          GroupLayout layout = new GroupLayout(panel);
89          panel.setLayout(layout);
90          layout.setAutoCreateGaps(autoCreatePadding:true);
91          layout.setAutoCreateContainerGaps(autoCreateContainerPadding:true);
92
93          JLabel sizeIdLabel = new JLabel(text:"Size ID:");
94          JTextField sizeIdText = new JTextField(columns:20);
95
96          JLabel nameLabel = new JLabel(text:"Brand:");
97          JTextField nameText = new JTextField(columns:20);
98
99          JLabel seasonTypeLabel = new JLabel(text:"Category (car/excavation/agriculture):"
100         JTextField seasonTypeText = new JTextField(columns:20);
101
102         JLabel salesRateLabel = new JLabel(text:"Sales Rate:");
103         JTextField salesRateText = new JTextField(columns:20);
104
105         JLabel expiringDateLabel = new JLabel(text:"Expiring Date (YYYY-MM-DD):");
106         JTextField expiringDateText = new JTextField(columns:20);
107
108         JLabel manufactureDateLabel = new JLabel(text:"Manufacture Date (YYYY-MM-DD):");
109         JTextField manufactureDateText = new JTextField(columns:20);
110
111         JLabel treadDepthLabel = new JLabel(text:"Tread Depth:");
112         JTextField treadDepthText = new JTextField(columns:20);
113
114         JLabel isFromFactoryLabel = new JLabel(text:"Is From Factory:");
115         JCheckBox isFromFactoryCheck = new JCheckBox();
```

The addTire method sets up a GUI form within a JPanel using GroupLayout, where it adds JLabels and corresponding JTextFields for various tire attributes (such as size ID, brand, category, sales rate, expiring date, manufacture date, tread depth, and factory status), along with a JCheckBox for factory status, to collect information from the user for adding a new tire to the inventory.

```
83      private static void addTire() {
119
120         JLabel originalPriceLabel = new JLabel(text:"Original Price:");
121         JTextField originalPriceText = new JTextField(columns:20);
122
123         JButton addButton = new JButton(text:"Add Tire");
124         addButton.addActionListener(new ActionListener() {
125             public void actionPerformed(ActionEvent e) {
126                 String sizeId = sizeIdText.getText();
127                 String name = nameText.getText();
128                 String seasonType = seasonTypeText.getText();
129                 int salesRate = Integer.parseInt(salesRateText.getText());
130                 LocalDate expiringDate = readDate(expiringDateText.getText());
131                 ArrayList<List<String>> warehouse = new ArrayList<>();
132                 LocalDate manufactureDate = readDate(manufactureDateText.getText());
133                 double treadDepth = Double.parseDouble(treadDepthText.getText());
134                 boolean isFromFactory = isFromFactoryCheck.isSelected();
135                 int numberOfTires = Integer.parseInt(numberOfTiresText.getText());
136                 double originalPrice = Double.parseDouble(originalPriceText.getText());
137
138                 Tire tire = new Tire(sizeId, name, seasonType, salesRate, expiringDate, warehouse, manufactureDate, treadDepth,
                    isFromFactory, numberOfTires, originalPrice);
139                 tires.add(tire);
140                 storage.storeTire(tire);
141                 JOptionPane.showMessageDialog(addFrame, message:"The tire has been successfully added and stored.");
142                 addFrame.dispose();
143             }
144         });
145
146         layout.setHorizontalGroup(
147                 layout.createSequentialGroup()
148                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
149                                 .addComponent(sizeIdLabel)
150                                 .addComponent(nameLabel)
151                                 .addComponent(seasonTypeLabel)
152                                 .addComponent(salesRateLabel)
153                                 .addComponent(expiringDateLabel)
154                                 .addComponent(manufactureDateLabel)
155                                 .addComponent(treadDepthLabel)
156                                 .addComponent(isFromFactoryLabel)
157                                 .addComponent(numberOfTiresLabel)
158                                 .addComponent(originalPriceLabel))
159                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING, resizable:false)
160                                 .addComponent(sizeIdText)
161                                 .addComponent(nameText)
```

The `addTire` method sets up a detailed form within a JPanel using GroupLayout, adding JLabels and corresponding JTextFields for various tire attributes (such as size ID, brand, category, sales rate, expiring date, manufacture date, tread depth, factory status, number of tires, and original price), and includes an add button that, when clicked, collects the input data, creates a new Tire object, adds it to the tire list and storage, and displays a success message to the user.

```
173         layout.setVerticalGroup(
174                 layout.createSequentialGroup()
175                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
176                                 .addComponent(sizeIdLabel)
177                                 .addComponent(sizeIdText))
178                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
179                                 .addComponent(nameLabel)
180                                 .addComponent(nameText))
181                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
182                                 .addComponent(seasonTypeLabel)
183                                 .addComponent(seasonTypeText))
184                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
185                                 .addComponent(salesRateLabel)
186                                 .addComponent(salesRateText))
187                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
188                                 .addComponent(expiringDateLabel)
189                                 .addComponent(expiringDateText))
190                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
191                                 .addComponent(manufactureDateLabel)
192                                 .addComponent(manufactureDateText))
193                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
194                                 .addComponent(treadDepthLabel)
195                                 .addComponent(treadDepthText))
196                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
197                                 .addComponent(isFromFactoryLabel)
198                                 .addComponent(isFromFactoryCheck))
199                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
200                                 .addComponent(numberOfTiresLabel)
201                                 .addComponent(numberOfTiresText))
202                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
203                                 .addComponent(originalPriceLabel)
204                                 .addComponent(originalPriceText))
205                         .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
206                                 .addComponent(addButton))
207         );
208 
209         addFrame.add(panel);
210         addFrame.setVisible(b:true);
211     }
```

The `addTire` method completes the setup of the form by defining the vertical group layout within GroupLayout, organizing all JLabels and JTextFields (including the size ID, brand, category, sales rate, expiring date, manufacture date, tread depth, factory status, number of tires, and original price) in parallel groups to ensure they align properly in the GUI, and finally, it adds the JPanel to the JFrame and makes the frame visible.

```
13  public class Main {
215     private static LocalDate readDate(String dateString) {
216         LocalDate date = null;
217         try {
218             date = LocalDate.parse(dateString, DateTimeFormatter.ISO_LOCAL_DATE);
219         } catch (DateTimeParseException e) {
220             JOptionPane.showMessageDialog(frame, message:"Invalid date format. Please try again.", title:"Error", JOptionPane.
                ERROR_MESSAGE);
221         }
222         return date;
223     }
224
225     private static void rankAndDisplayTires() {
226         TireDSS tireDSS = new TireDSS();
227         tires.sort((t1, t2) -> tireDSS.calculateScore(t2).getTotalScore() - tireDSS.calculateScore(t1).getTotalScore());
228
229         StringBuilder ranking = new StringBuilder();
230         for (Tire tire : tires) {
231             DetailedScore score = tireDSS.calculateScore(tire);
232             if (score.getTotalScore() > 0) {
233                 ranking.append(str:"Tire: ").append(tire.getBrand()).append(str:" | Total Score: ").append(score.getTotalScore()).append
                    (str:"\n")
234                         .append(str:"  Sales Rate Contribution: ").append(score.getSalesRateContribution()).append(str:"\n")
235                         .append(str:"  Sales Rate and Low Tire Count Contribution: ").append(score.getSalesRateAndLowTireCountContribution
                            ()).append(str:"\n")
236                         .append(str:"  Sales Rate, Low Tire Count and Expiry Contribution: ").append(score.
                            getSalesRateLowTireCountExpiryContribution()).append(str:"\n")
237                         .append(str:"  Low Tire Count Contribution: ").append(score.getLowTireCountContribution()).append(str:"\n")
238                         .append(str:"  Low Tire Count and Expiry Contribution: ").append(score.getLowTireCountAndExpiryContribution()).
                            append(str:"\n")
239                         .append(str:"  Expiry Soon Contribution: ").append(score.getExpirySoonContribution()).append(str:"\n");
240
241                 if (score.getTotalScore() > 0) {
242                     ranking.append(str:"  Size ID: ").append(tire.getSizeId()).append(str:"\n")
243                             .append(str:"  Brand: ").append(tire.getBrand()).append(str:"\n")
244                             .append(str:"  Category: ").append(tire.getCategory()).append(str:"\n")
245                             .append(str:"  Sales Rate: ").append(tire.getSalesRate()).append(str:"\n")
246                             .append(str:"  Expiring Date: ").append(tire.getExpiringDate()).append(str:"\n")
247                             .append(str:"  Manufacturing Date: ").append(tire.getManufactureDate()).append(str:"\n")
248                             .append(str:"  Tread Depth: ").append(tire.getTreadDepth()).append(str:"\n")
249                             .append(str:"  Is From Factory: ").append(tire.isFromFactory()).append(str:"\n")
250                             .append(str:"  Number of Tires: ").append(tire.getNumberOfTires()).append(str:"\n");
251                 }
252                 ranking.append(str:"\n");
253             }
254         }
```

4.8 Detailed Explanation of `readDate` and `rankAndDisplayTires` Methods

The `readDate` method attempts to parse a date string into a `LocalDate` object using the `ISO_LOCAL_DATE` format. If parsing fails, it catches the `DateTimeParseException` and displays an error message using `JOptionPane`, informing the user of the invalid date format. The method then returns the parsed date or null if parsing was unsuccessful.

The `rankAndDisplayTires` method sorts the list of tires based on their total score, calculated by a `TireDSS` object. It initializes a `StringBuilder` named `ranking` to collect information about the ranked tires. For each tire, it calculates a detailed score and appends various contributions to the total score, including sales rate, low tire count, expiry, and other factors, to the `ranking` `StringBuilder`. If the total score of the tire is greater than zero, the method also appends detailed information about the tire, such as its size ID, brand, category, sales rate, expiring date, manufacturing date, tread depth, factory status, and the number of tires. Finally, the method appends the complete ranking information to the `ranking` `StringBuilder` and likely displays it in some form, although the display code is not shown in the provided snippet.

Together, these methods enhance the tire inventory management system by allowing users to apply discounts based on sales rates, handle date input robustly, and rank tires based on various factors, providing a comprehensive overview of tire inventory and performance.

```
264
265     private static void checkOrderRequirements() {
266         StringBuilder orderRequirements = new StringBuilder();
267         for (Tire tire : tires) {
268             if (tire.getNumberOfTires() < 12) {
269                 orderRequirements.append(str:"Order required for ").append(tire.getSizeId() + " " + tire.getBrand()).append(str:" tires.
                    \n");
270                 StringBuilder orderRequirements - org.example.Main.checkOrderRequirements()
271             }
272             if ( orderRequirements
273                 orderRequirements.append(str:"No orders required.");
274             }
275             JOptionPane.showMessageDialog(frame, orderRequirements.toString(), title:"Order Requirements", JOptionPane.INFORMATION_MESSAGE);
276     }
277
278     private static void applyAndDisplayDiscounts() {
279         StringBuilder discountedTires = new StringBuilder(str:"Discounted Tires:\n");
280         for (Tire tire : tires) {
281             if (tire.getSalesRate() <= 40) {
282                 double discountedPrice = tire.getOriginalPrice() * 0.9; // Applying 10% discount for low sales rate
283                 tire.setDiscountedPrice(discountedPrice);
284                 discountedTires.append(str:"Tire: ").append(tire.getBrand()).append(str:" | Original Price: ").append(tire.getOriginalPrice
                    ()).append(str:" | Suggested Discounted Price: ").append(tire.getDiscountedPrice()).append(str:"\n");
285             }
286         }
287         JOptionPane.showMessageDialog(frame, discountedTires.toString());
288     }
```

The checkOrderRequirements method is designed to evaluate the inventory levels of tires and determine if new orders are needed. It initializes a StringBuilder called orderRequirements to collect information about the tires that require reordering. The method iterates through the list of tires, checking if the number of tires for each tire type is less than 12. If this condition is met, it appends a message to the orderRequirements StringBuilder, indicating the size ID and brand of the tire that needs to be ordered. After the loop, if no tires meet the reorder condition, it appends a message stating "No orders required." Finally, the method displays the collected information in a message dialog using JOptionPane, providing the user with a clear summary of the order requirements.

The applyAndDisplayDiscounts method focuses on identifying tires with low sales rates and applying a discount to them. It starts by initializing a StringBuilder named discountedTires to store information about the discounted tires. The method iterates through the list of tires, checking if the sales rate of each tire is less than or equal to 40. For those tires, it calculates a discounted price by applying a 10% reduction to the original price and updates the tire's price accordingly. The method then appends the brand, original price, and the new discounted price of each discounted tire to the discountedTires StringBuilder. At the end of the loop, it displays the collected discount information in a message dialog using JOptionPane, informing the user about the tires that received a discount and their new prices.

Together, these methods enhance the functionality of the tire inventory management system by providing essential features for maintaining optimal inventory levels and applying strategic discounts to improve sales performance.

```
290     private static void displayWarehouses() {
291         StringBuilder warehouseContents = new StringBuilder();
292         warehouseContents.append(str:"New Car Warehouse:\n");
293         storage.getCarNewWarehouse().getAllTires().forEach((brand, tires) -> {
294             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
295             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
296         });
297
298         warehouseContents.append(str:"\nOld Car Warehouse:\n");
299         storage.getCarOldWarehouse().getAllTires().forEach((brand, tires) -> {
300             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
301             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
302         });
303
304         warehouseContents.append(str:"\nNew Excavation Warehouse:\n");
305         storage.getExcavationNewWarehouse().getAllTires().forEach((brand, tires) -> {
306             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
307             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
308         });
309
310         warehouseContents.append(str:"\nOld Excavation Warehouse:\n");
311         storage.getExcavationOldWarehouse().getAllTires().forEach((brand, tires) -> {
312             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
313             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
314         });
315
316         warehouseContents.append(str:"\nNew Agriculture Warehouse:\n");
317         storage.getAgriculturalNewWarehouse().getAllTires().forEach((brand, tires) -> {
318             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
319             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
320         });
321
322         warehouseContents.append(str:"\nOld Agriculture Warehouse:\n");
323         storage.getAgriculturalOldWarehouse().getAllTires().forEach((brand, tires) -> {
324             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
```

The `displayWarehouses` method generates a detailed overview of the contents of each warehouse by iterating through the stored tires and collecting relevant information. It initializes a `StringBuilder` named `warehouseContents` to accumulate the data to be displayed. For each warehouse, such as "New Car Warehouse," "Old Car Warehouse," "New Excavation Warehouse," "Old Excavation Warehouse," "New Agriculture Warehouse," and "Old Agriculture Warehouse," the method retrieves the list of tires and iterates over each tire's brand and attributes. During each iteration, it appends the brand, size ID, manufacturing date, and the number of tires to the `warehouseContents` `StringBuilder`. This detailed information is formatted and stored for each warehouse, providing a comprehensive inventory listing. After aggregating all the data, the method likely displays the `warehouseContents` in a message dialog or another GUI component, allowing the user to view a complete and organized summary of the tire inventory across all warehouses. This method ensures that users can quickly and easily access detailed information about their tire stock, facilitating better inventory management and decision-making.

```
290     private static void displayWarehouses() {
291         StringBuilder warehouseContents = new StringBuilder();
292         warehouseContents.append(str:"New Car Warehouse:\n");
293         storage.getCarNewWarehouse().getAllTires().forEach((brand, tires) -> {
294             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
295             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
296         });
297
298         warehouseContents.append(str:"\nOld Car Warehouse:\n");
299         storage.getCarOldWarehouse().getAllTires().forEach((brand, tires) -> {
300             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
301             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
302         });
303
304         warehouseContents.append(str:"\nNew Excavation Warehouse:\n");
305         storage.getExcavationNewWarehouse().getAllTires().forEach((brand, tires) -> {
306             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
307             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
308         });
309
310         warehouseContents.append(str:"\nOld Excavation Warehouse:\n");
311         storage.getExcavationOldWarehouse().getAllTires().forEach((brand, tires) -> {
312             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
313             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
314         });
315
316         warehouseContents.append(str:"\nNew Agriculture Warehouse:\n");
317         storage.getAgriculturalNewWarehouse().getAllTires().forEach((brand, tires) -> {
318             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
319             tires.forEach(tire -> warehouseContents.append(str:" - The size Id: ").append(tire.getSizeId()).append(str:" & Manufacturing
                Date: ").append(tire.getManufactureDate()).append(str:" & Number of Tires: ").append(tire.getNumberOfTires()).append
                (str:"\n"));
320         });
321
322         warehouseContents.append(str:"\nOld Agriculture Warehouse:\n");
323         storage.getAgriculturalOldWarehouse().getAllTires().forEach((brand, tires) -> {
324             warehouseContents.append(str:"Brand: ").append(brand).append(str:"\n");
```

The `displayWarehouses` method is designed to generate and display a detailed summary of the contents of each warehouse by iterating through the tires stored in them. It starts by initializing a `StringBuilder` named `warehouseContents` to accumulate the information that will be displayed. For each warehouse—such as "New Car Warehouse," "Old Car Warehouse," "New Excavation Warehouse," "Old Excavation Warehouse," "New Agriculture Warehouse," and "Old Agriculture Warehouse"—the method retrieves the list of tires stored in that warehouse using the `getAllTires()` method of the `Storage` class.

Within each warehouse block, the method appends a header to the `warehouseContents` indicating the name of the warehouse. It then iterates over the retrieved list of tires using a lambda function. For each tire, the method appends the brand, size ID, manufacturing date, and the number of tires to the `warehouseContents` `StringBuilder`. This detailed information is formatted to include labels and newlines to ensure readability.

For example, in the block for "New Car Warehouse," the method calls `storage.getCarNewWarehouse().getAllTires().forEach((brand, tires) -> {...})` to retrieve and iterate over the tires. Inside the lambda function, it appends the tire details, including the brand, size ID, manufacturing date, and number of tires, to the `warehouseContents`.

This process is repeated for each warehouse, ensuring that the contents of all warehouses are thoroughly documented. The use of a `StringBuilder` allows for efficient accumulation of the information, and appending newlines ensures that each tire's information is on a separate line for clarity.

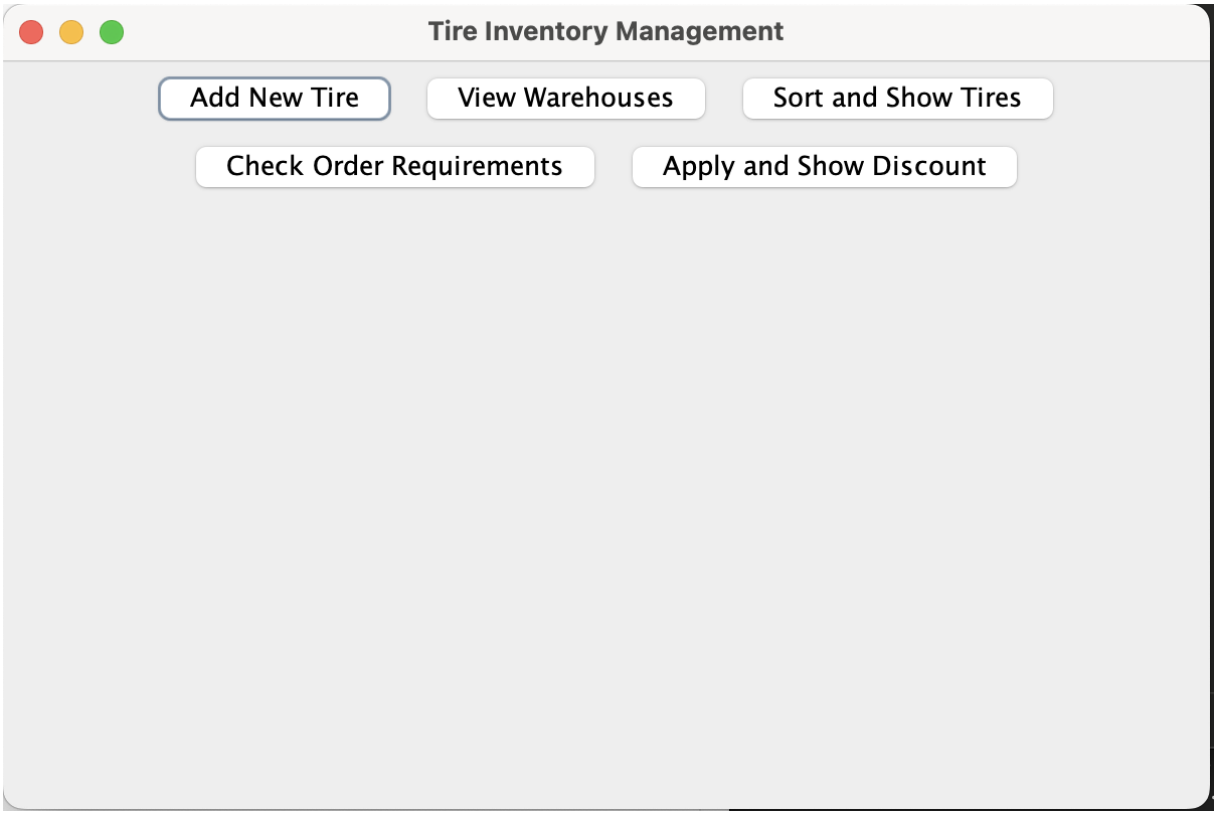
Finally, after accumulating the data for all warehouses, the method likely uses a `JOptionPane` or similar GUI component to display the contents of the `warehouseContents` `StringBuilder` to the user. This detailed summary allows the user to easily view and manage the tire inventory across all warehouses, facilitating better inventory control and decision-making. The method ensures that users can access a comprehensive overview of the tire stock, including critical details for each tire, all formatted in an organized and readable manner.

4.9 Conclusion of Main Method

The `main` method serves as the entry point for the tire inventory management application, initializing several `Warehouse` objects for different tire categories (new car, old car, new excavation, old excavation, new agriculture, and old agriculture), and consolidating them under a single `Storage` object to manage these warehouses efficiently. It then calls the `addInitialData()` method to populate the system with initial data, ensuring that the application starts with a baseline dataset for testing and demonstration purposes. Finally, the `createMainMenu()` method is invoked to set up the main graphical user interface, which includes buttons for adding new tires, viewing warehouses, sorting and displaying tires, checking order requirements, and applying discounts. Each button is linked to an action listener, providing an interactive and user-friendly interface for managing the tire inventory effectively.

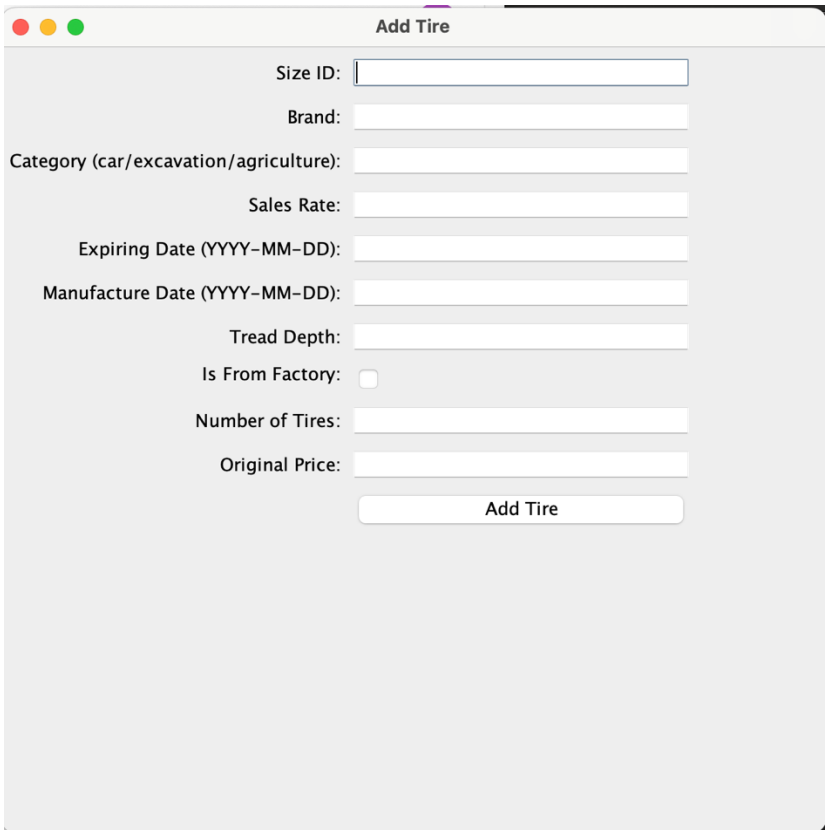
5 Interface Design

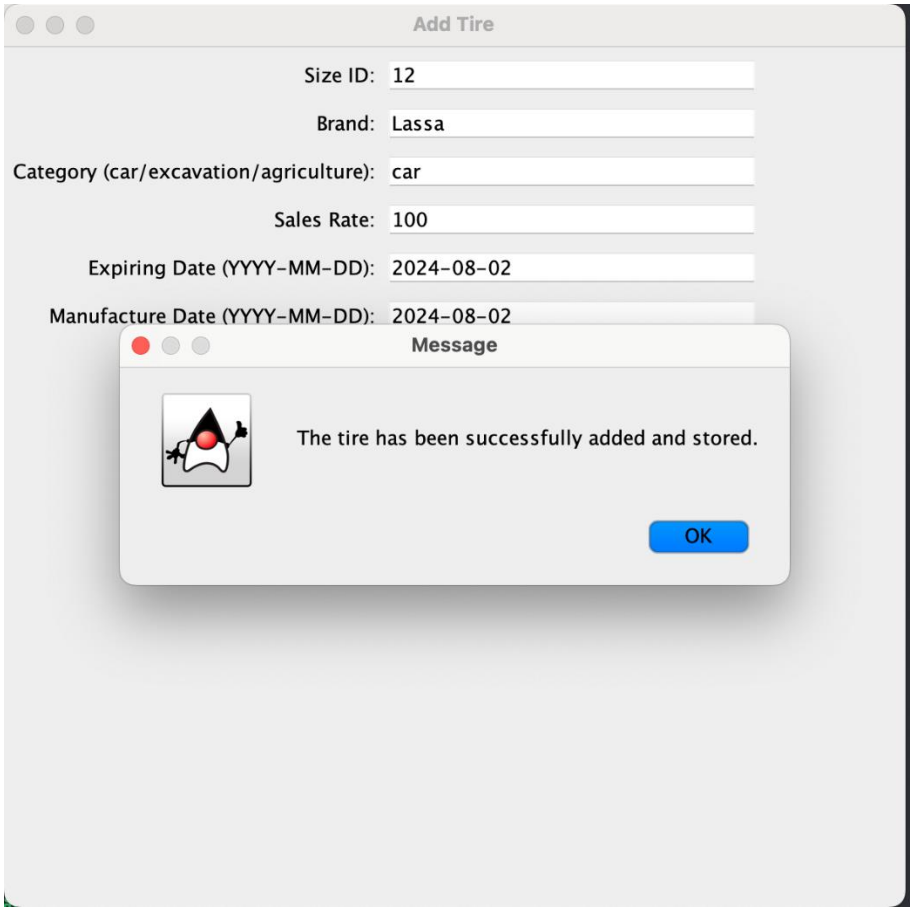
5.1. Main Monitor



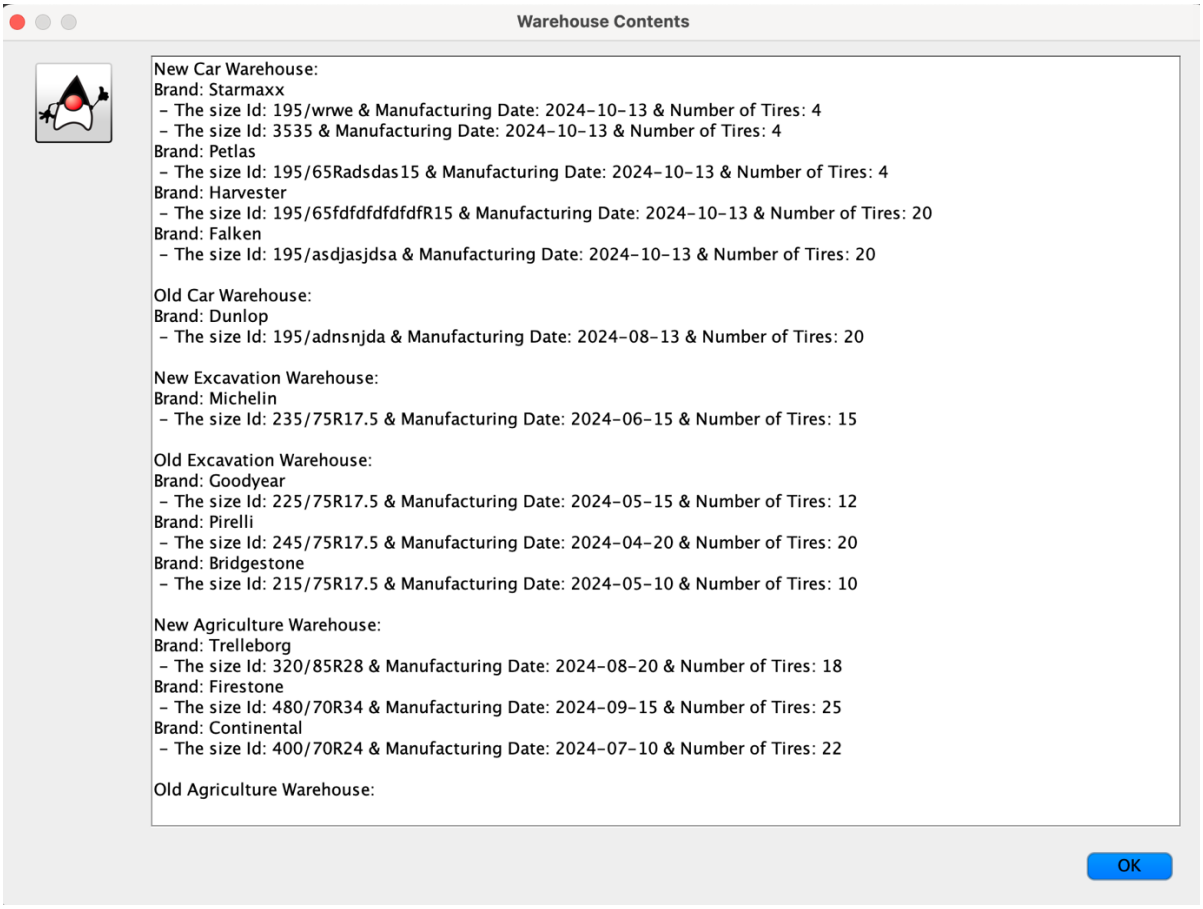
The user interface shown in the image is the main menu of the Tire Inventory Management application. It features a simple and clean layout with five primary buttons, each providing access to a key functionality of the application:

1. **Add New Tire**: This button allows users to add a new tire to the inventory. Clicking this button likely opens a form where the user can input various details about the tire, such as its size, brand, category, sales rate, expiration date, manufacture date, tread depth, and other relevant information.

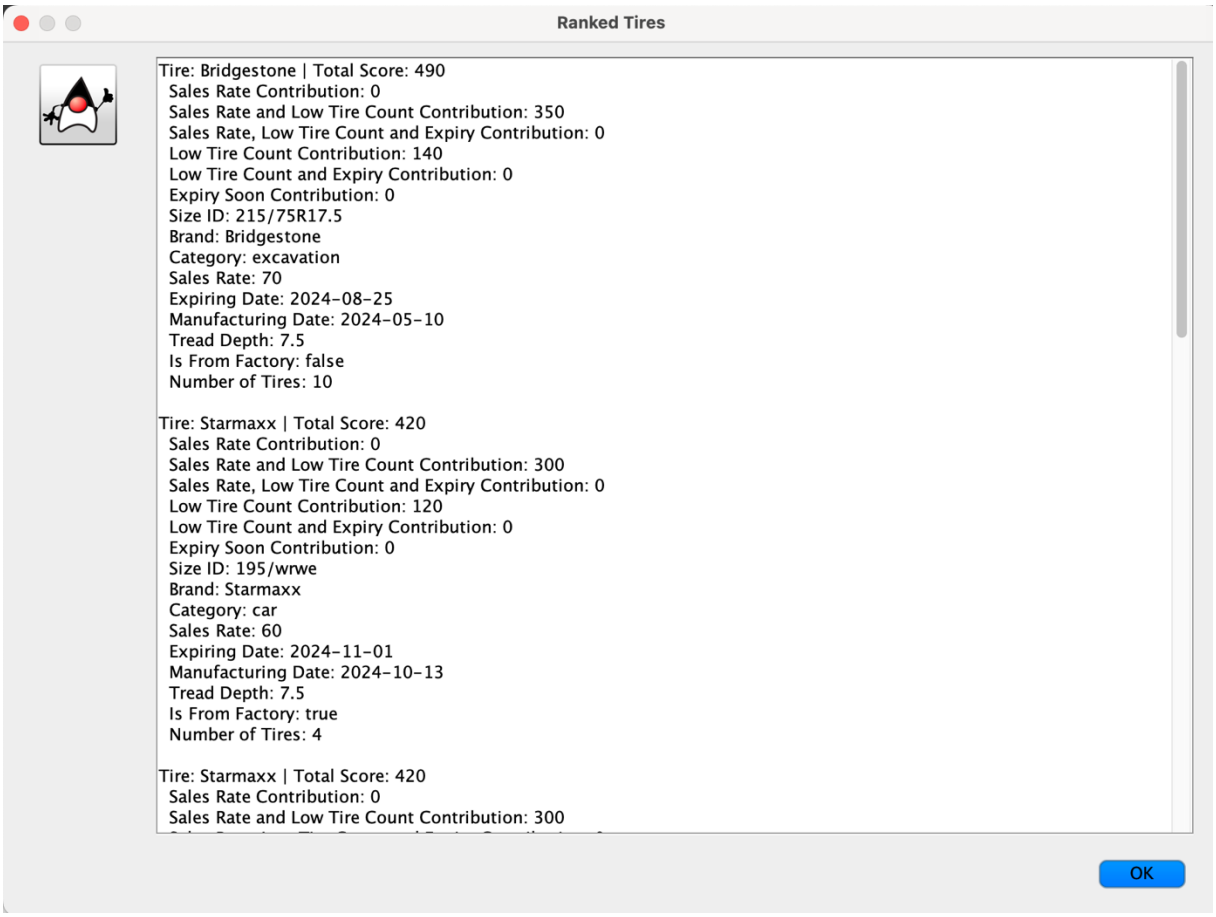

Add Tire
Size ID:
Brand:
Category (car/excavation/agriculture):
Sales Rate:
Expiring Date (YYYY-MM-DD):
Manufacture Date (YYYY-MM-DD):
Tread Depth:
Is From Factory:
Number of Tires:
Original Price:
Add Tire


Add Tire
Size ID: 12
Brand: Lassa
Category (car/excavation/agriculture): car
Sales Rate: 100
Expiring Date (YYYY-MM-DD): 2024-08-02
Manufacture Date (YYYY-MM-DD): 2024-08-02
Message
The tire has been successfully added and stored.
OK

2. **View Warehouses:** This button provides users with an overview of the different warehouses and their contents. It likely displays a list or a detailed view of the tires stored in each warehouse, helping users to manage and track inventory.

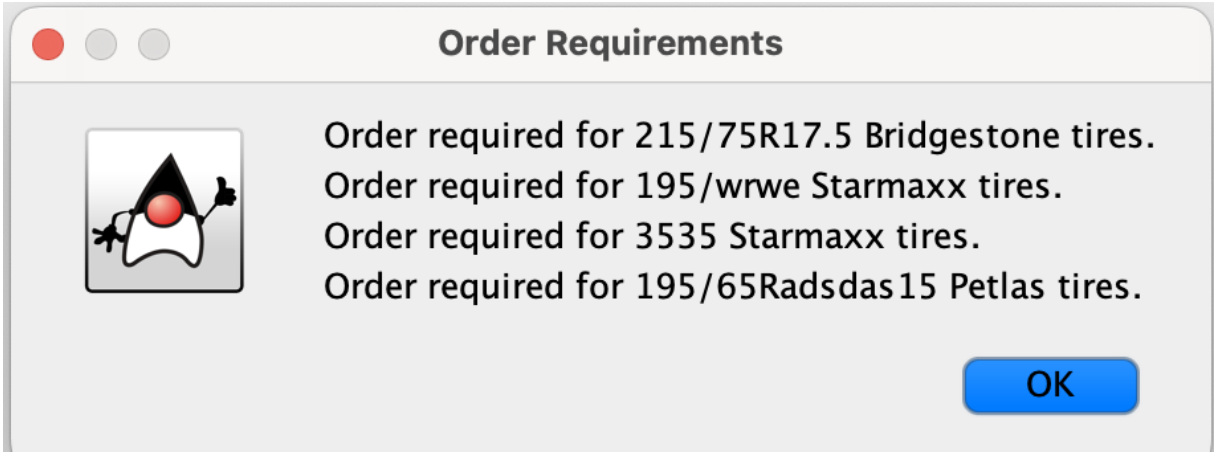


3. **Sort and Show Tires:** Clicking this button probably allows users to sort the tires based on different criteria such as size, brand, category, or sales rate. This helps users to easily find and organize tires according to their needs.



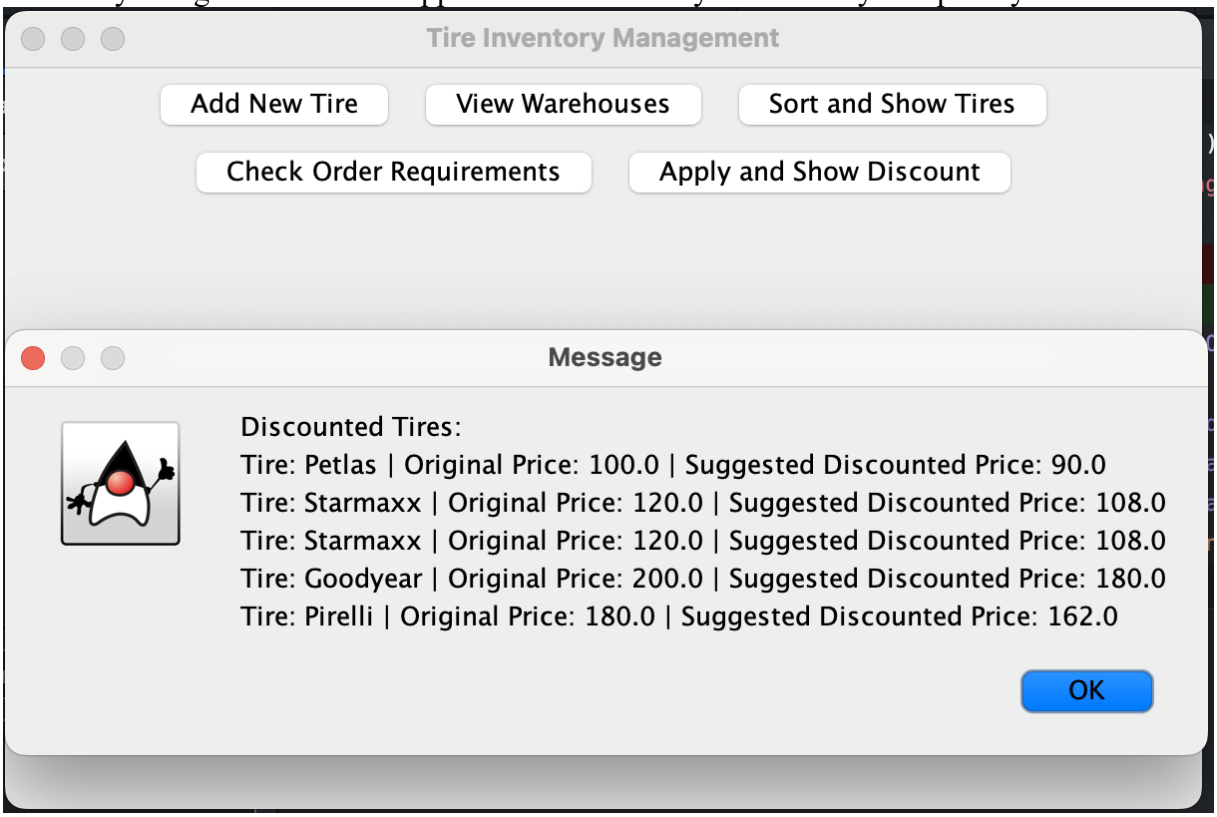
The ranking system evaluates and scores tires based on multiple criteria, resulting in a total score that reflects the overall performance or importance of each tire. Each tire's total score is the sum of several individual contributions, each assessing different aspects of the tire's attributes. The Sales Rate Contribution measures how well a tire is selling, although in the provided examples, this contribution is zero, possibly indicating it is not a significant factor for these tires. The Sales Rate and Low Tire Count Contribution assesses the tire's sales performance in conjunction with its stock level, where a low stock level coupled with high sales can indicate high demand and thus contribute positively to the total score, as seen with the Bridgestone tire's 350-point contribution. The Sales Rate, Low Tire Count, and Expiry Contribution combines the tire's sales rate, stock level, and proximity to expiry, reflecting the urgency and demand for the tire. The Low Tire Count Contribution focuses solely on the stock level, where a low number of tires in inventory contributes positively, indicating a need for restocking. The Expiry Soon Contribution measures how close a tire is to its expiry date, contributing to the total score based on the urgency of selling the tire before it expires. Additional attributes such as Size ID,Brand, Category, Expiring Date, Manufacturing Date,Tread Depth, Factory Status, and Number of Tires provide detailed information that supports the scoring process, helping users understand the factors influencing each tire's score and overall ranking within the inventory. This comprehensive scoring system aids in prioritizing inventory management decisions by highlighting the most critical tires based on multiple performance metrics.

4. Check Order Requirements: This button helps users to check the current inventory levels and determine if new orders are required. It likely runs a check against predefined thresholds and alerts users if any tires need to be reordered.



5. Apply and Show Discount:

Each button is designed to perform a specific action, making the interface user-friendly and efficient for managing the tire inventory. The clean and minimalistic design ensures that users can easily navigate and use the application without any unnecessary complexity.



The `applyAndDisplayDiscounts` method applies a 10% discount to tires that have a sales rate of 60 or less. It begins by initializing a `StringBuilder` named `discountedTires` with the header "Discounted Tires:\n" to accumulate details about the tires that qualify for the discount. The method then iterates through the list of tire objects. For each tire, it checks if the tire's sales rate, retrieved using the `getSalesRate()` method, is less than or equal to 60. If the condition is met, the method retrieves the tire's original price, calculates the discounted price by applying a 10%

reduction, and sets the new discounted price using the `setDiscountedPrice(discountedPrice)` method. The method then appends detailed information about the discounted tire to the `discountedTires` `StringBuilder`, including the tire's brand, original price, and the new discounted price. After processing all the tires, the method uses a `JOptionPane` to display the accumulated information in a message dialog, providing a summary of all tires that received the discount. For example, if a tire from the brand Michelin with a sales rate of 55 and an original price of $200 qualifies, its discounted price will be $180, and this information will be included in the displayed message. Similarly, another tire from the brand Bridgestone with a sales rate of 60 and an original price of $180 will also be discounted to $162 and displayed in the summary. This method ensures that tires with lower sales rates are discounted and provides users with a clear overview of the discounted products.

6 Testing Process and Results

6.1 Storage Test:

Scenario 1 : testStoreTire

Objective: Test storing a tire in the appropriate warehouse when the tire is new and suitable for a new warehouse.

Setup:

- Six Warehouse objects are created.
- A Tire object is created.

Execution: The storeTire method of the Storage object is called with the Tire object.
Assertion: The test checks if the newCarWarehouse now contains 1 tire.

```
9   public class StorageTest {
10
11      @Test
12      public void testStoreTire() {
13          Warehouse newCarWarehouse = new Warehouse("New Car Warehouse");
14          Warehouse oldCarWarehouse = new Warehouse("Old Car Warehouse");
15          Warehouse newExcavationWarehouse = new Warehouse("New Excavation Warehouse");
16          Warehouse oldExcavationWarehouse = new Warehouse("Old Excavation Warehouse");
17          Warehouse newAgricultureWarehouse = new Warehouse("New Agriculture Warehouse");
18          Warehouse oldAgricultureWarehouse = new Warehouse("Old Agriculture Warehouse");
19
20          Storage storage = new Storage(newCarWarehouse, oldCarWarehouse, newExcavationWarehouse, oldExcavationWarehouse, newAgricultureWarehouse, oldAgricultureWarehouse);
21
22          ArrayList<List<String>> warehouse = new ArrayList<>();
23          Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10, 13), 7.5, true, 4, 100.0);
24          storage.storeTire(tire);
25
26          assertEquals(1, storage.getCarNewWarehouse().getTires().size());
27      }
```

Test Explanation : The tire is new (isNew: true) and is intended for a car (type: "car"). Based on these attributes, the tire should be stored in the newCarWarehouse. The test ensures that after calling storeTire, the newCarWarehouse has exactly one tire.

Scenario 2: testStoreTireToOldWarehouse

Objective: Test storing a tire in the appropriate warehouse when the tire is old and suitable for an old warehouse.

Setup:

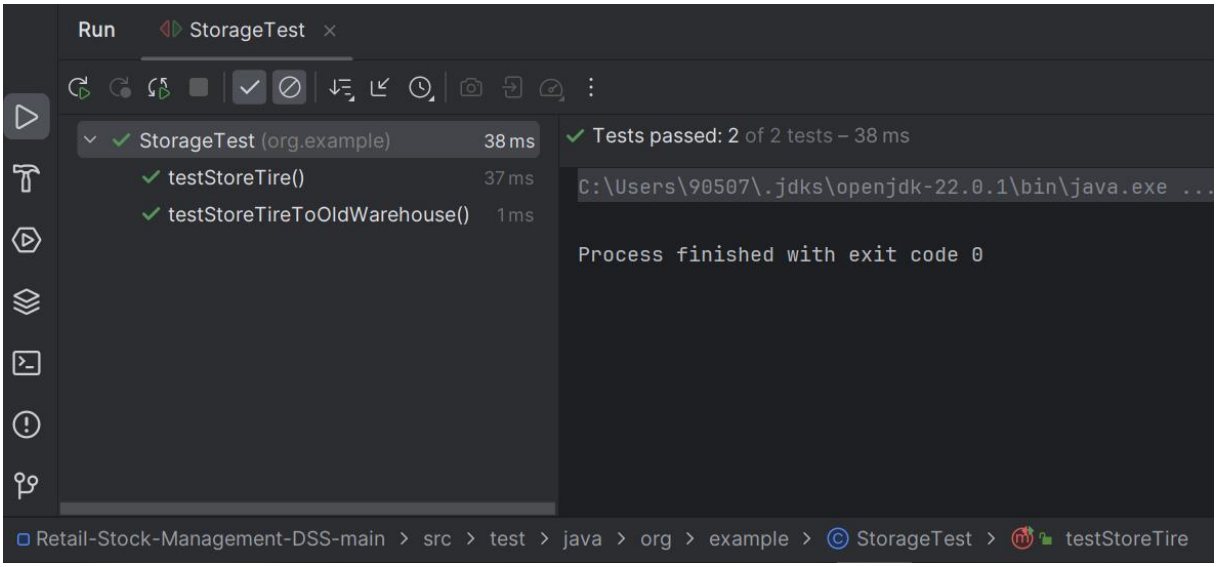
- The same six Warehouse objects are created.
- Another Storage object is instantiated with these warehouses.
- A Tire object is created.

Execution: The storeTire method of the Storage object is called with the Tire object.
Assertion: The test checks if the `oldCarWarehouse` now contains 1 tire.

```java
@Test
public void testStoreTireToOldWarehouse() {
    Warehouse newCarWarehouse = new Warehouse("New Car Warehouse");
    Warehouse oldCarWarehouse = new Warehouse("Old Car Warehouse");
    Warehouse newExcavationWarehouse = new Warehouse("New Excavation Warehouse");
    Warehouse oldExcavationWarehouse = new Warehouse("Old Excavation Warehouse");
    Warehouse newAgricultureWarehouse = new Warehouse("New Agriculture Warehouse");
    Warehouse oldAgricultureWarehouse = new Warehouse("Old Agriculture Warehouse");

    Storage storage = new Storage(newCarWarehouse, oldCarWarehouse, newExcavationWarehouse, oldExcavationWarehouse, newAgricultureWarehouse, oldAgricultureWarehouse);

    ArrayList<List<String>> warehouse = new ArrayList<>();
    Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2023, 10, 13), 6.5, false, 4, 100.0);
    storage.storeTire(tire);

    assertEquals(1, storage.getCarOldWarehouse().getTires().size());
}
}
```

Test Explanation : The tire is not new (isNew: false) and is intended for a car (type: "car"). Based on these attributes, the tire should be stored in the oldCarWarehouse. The test ensures that after calling storeTire, the oldCarWarehouse has exactly one tire.

Output:



Summary of Storage Test:
Both tests validate that the storeTire method correctly assigns tires to the appropriate warehouse based on their attributes:

- New car tires go to newCarWarehouse.
- Old car tires go to oldCarWarehouse.

By using different isNew values and checking the corresponding warehouse, the tests cover both scenarios of tire storage.

6.2 TireDSSTest

Scenario 1: testCalculateScore

Objective : Test the calculation of the score for a tire with high sales rate and high remaining tread depth.
Setup:

- A Tire object is created.

Execution: The calculateScore method of the TireDSS class is called with the Tire object.

Assertion: The test checks if the score components are calculated as follows:

- SalesRateAndLowTireCountContribution: 300
- SalesRateLowTireCountExpiryContribution: 0
- LowTireCountContribution: 120
- LowTireCountAndExpiryContribution: 0
- ExpirySoonContribution: 0

```
9   public class TireDSSTest {
10
11      @Test
12      public void testCalculateScore() {
13          ArrayList<List<String>> warehouse = new ArrayList<>();
14          Tire tire = new Tire("195/65R15", "Petlas", "car", 60, LocalDate.of(2024, 11, 13), warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 8, 100.0);
15          TireDSS tireDSS = new TireDSS();
16          DetailedScore score = tireDSS.calculateScore(tire);
17
18          assertEquals(300, score.getSalesRateAndLowTireCountContribution());
19          assertEquals(0, score.getSalesRateLowTireCountExpiryContribution());
20          assertEquals(120, score.getLowTireCountContribution());
21          assertEquals(0, score.getLowTireCountAndExpiryContribution());
22          assertEquals(0, score.getExpirySoonContribution());
23      }
```

Test Explanation: The tire has a high sales rate (60) and a high tread depth (7.5), contributing to high scores in the related components.

Scenario 2: testLowTireCountScore

Objective : Test the score calculation for a tire with a lower sales rate and still a considerable amount of tread.
Setup:
A Tire object is created with:

- Sales rate: 40
- Other attributes are similar to the first test case.

Execution: The calculateScore method is called with the Tire object.
Assertion: The test checks if the score components are calculated as follows:

- SalesRateContribution: 0

- SalesRateAndLowTireCountContribution: 0
- SalesRateLowTireCountExpiryContribution: 0
- LowTireCountContribution: 80
- LowTireCountAndExpiryContribution: 0
- ExpirySoonContribution: 0

```
25      @Test
26      public void testLowTireCountScore() {
27          ArrayList<List<String>> warehouse = new ArrayList<>();
28          Tire tire = new Tire("195/65R15", "Petlas", "car", 40, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 8, 100.0);
29          TireDSS tireDSS = new TireDSS();
30          DetailedScore score = tireDSS.calculateScore(tire);
31
32          assertEquals(0, score.getSalesRateContribution());
33          assertEquals(0, score.getSalesRateAndLowTireCountContribution());
34          assertEquals(0, score.getSalesRateLowTireCountExpiryContribution());
35          assertEquals(80, score.getLowTireCountContribution());
36          assertEquals(0, score.getLowTireCountAndExpiryContribution());
37          assertEquals(0, score.getExpirySoonContribution());
38      }
```

Test Explanation: The lower sales rate (40) results in no contribution from the sales rate components. The remaining tread still contributes to the low tire count component.

Scenario 3: testLowTireAndHighSalesScore

Objective: Test the score calculation for a tire with a high sales rate and a high remaining tread depth.
Setup:

- A Tire object is created with:
  - Sales rate: 70
  - Other attributes are similar to the first test case.

Execution: The calculateScore method is called with the Tire object.
Assertion: The test checks if the score components are calculated as follows:

- SalesRateContribution: 0
- SalesRateAndLowTireCountContribution: 350
- SalesRateLowTireCountExpiryContribution: 0
- LowTireCountContribution: 140
- LowTireCountAndExpiryContribution: 0
- ExpirySoonContribution: 0

```
40      @Test
41      public void testLowTireAndHighSalesScore() {
42          ArrayList<List<String>> warehouse = new ArrayList<>();
43          Tire tire = new Tire("195/65R15", "Petlas", "car", 70, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 10, 100.0);
44          TireDSS tireDSS = new TireDSS();
45          DetailedScore score = tireDSS.calculateScore(tire);
46
47          assertEquals(0, score.getSalesRateContribution());
48          assertEquals(350, score.getSalesRateAndLowTireCountContribution());
49          assertEquals(0, score.getSalesRateLowTireCountExpiryContribution());
50          assertEquals(140, score.getLowTireCountContribution());
51          assertEquals(0, score.getLowTireCountAndExpiryContribution());
52          assertEquals(0, score.getExpirySoonContribution());
53      }
```

Test Explanation: The higher sales rate (70) and high tread depth (10) result in significant contributions from the related components.

Scenario 4 : testCalculateScoreForEverySituation

Objective : Test the score calculation for a tire considering sales rate, low tire count, and soon expiry.
Setup:

- A Tire object is created with:
  - Sales rate: 70
  - Expiry date soon (10 days from now)
  - Other attributes are similar to the previous cases.

Execution: The calculateScore method is called with the Tire object.
Assertion: The test checks if the score components are calculated as follows:

- SalesRateAndLowTireCountContribution: 350
- SalesRateLowTireCountExpiryContribution: 280
- LowTireCountContribution: 140
- LowTireCountAndExpiryContribution: 210
- ExpirySoonContribution: 70

```
55      @Test
56      public void testCalculateScoreForEverySituation() {
57          ArrayList<List<String>> warehouse = new ArrayList<>();
58          Tire tire = new Tire("195/65R15", "Petlas", "car", 70, LocalDate.now().plusDays(10),  warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 10, 100.0);
59          TireDSS tireDSS = new TireDSS();
60          DetailedScore score = tireDSS.calculateScore(tire);
61
62          assertEquals(350, score.getSalesRateAndLowTireCountContribution());
63          assertEquals(280, score.getSalesRateLowTireCountExpiryContribution());
64          assertEquals(140, score.getLowTireCountContribution());
65          assertEquals(210, score.getLowTireCountAndExpiryContribution());
66          assertEquals(70, score.getExpirySoonContribution());
67      }
```

Test Explanation : The tire has a high sales rate, high tread depth, and is expiring soon. All these factors contribute to the respective components of the score.

Scenario 5 : testCalculateScoreForExpirySoon

Objective: Test the score calculation for a tire that is expiring soon without considering other factors.
Setup:

- A Tire object is created with:
  - Expiry date soon (10 days from now)
  - High sales rate (70)
  - Tread depth: 40
  - Other attributes are similar to previous cases.

Execution: The calculateScore method is called with the Tire object.
Assertion: The test checks if the score components are calculated.

```
69      @Test
70      public void testCalculateScoreForExpirySoon() {
71          ArrayList<List<String>> warehouse = new ArrayList<>();
72          Tire tire = new Tire("195/65R15", "Petlas", "car", 70, LocalDate.now().plusDays(10),  warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 40, 100.0);
73          TireDSS tireDSS = new TireDSS();
74          DetailedScore score = tireDSS.calculateScore(tire);
75
76          assertEquals(0, score.getSalesRateContribution());
77          assertEquals(0, score.getSalesRateAndLowTireCountContribution());
78          assertEquals(0, score.getSalesRateLowTireCountExpiryContribution());
79          assertEquals(0, score.getLowTireCountContribution());
80          assertEquals(0, score.getLowTireCountAndExpiryContribution());
81          assertEquals(70, score.getExpirySoonContribution());
82      }
```

Test Explanation: The tire is primarily expiring soon, which contributes to the expiry soon component.

Scenario 6: testCalculateScoreForLowTireCountAndExpirySoon

Objective : Test the score calculation for a tire with low tire count and expiring soon

Setup:

- A Tire object is created with:
    - Sales rate: 50
    - Expiry date soon (10 days from now)
    - Tread depth: 5
    - Other attributes are similar to previous cases.

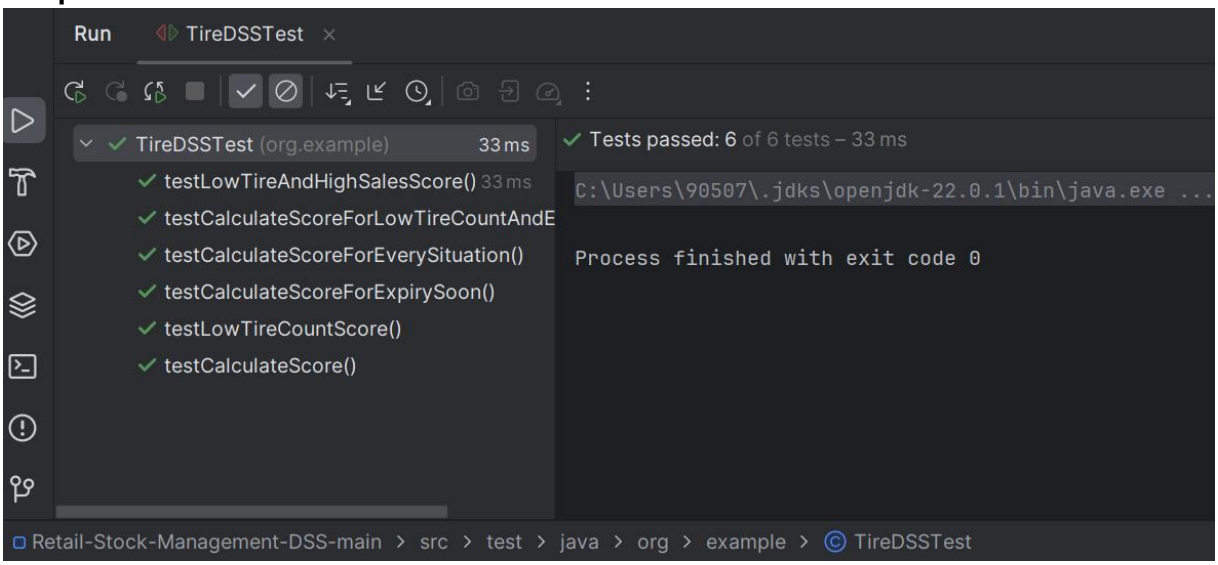
Execution: The calculateScore method is called with the Tire object.

Assertion: The test checks if the score components are calculated

```
84      @Test
85      public void testCalculateScoreForLowTireCountAndExpirySoon() {
86          ArrayList<List<String>> warehouse = new ArrayList<>();
87          Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.now().plusDays(10),  warehouse, LocalDate.of(2024, 10,
              13), 8.0, true, 5, 100.0);
88          TireDSS tireDSS = new TireDSS();
89          DetailedScore score = tireDSS.calculateScore(tire);
90
91          assertEquals(0, score.getSalesRateContribution());
92          assertEquals(0, score.getSalesRateAndLowTireCountContribution());
93          assertEquals(0, score.getSalesRateLowTireCountExpiryContribution());
94          assertEquals(100, score.getLowTireCountContribution());
95          assertEquals(150, score.getLowTireCountAndExpiryContribution());
96          assertEquals(50, score.getExpirySoonContribution());
97      }
98  }
```

Test Explanation : The tire has a lower tire count and is expiring soon, contributing to the respective components in the score calculation.

Output:



Summary of Score Test:

These test cases cover a range of scenarios to ensure the TireDSS class calculates the detailed score accurately based on various attributes of the tires, such as sales rate, remaining tread depth, and expiry date. Each test case isolates specific conditions to validate the scoring logic comprehensively.

6.3 Tire Test

Scenario 1: testTireCreation

Objective: Ensure that a Tire object is created correctly with the specified properties.

Setup:

- A Tire object is created with specific attributes.
- Attributes include size, brand, category, sales rate, expiring date, manufacture date, tread depth, factory status, number of tires, and original price.

Execution and Assertion:

- The test verifies that each attribute of the Tire object matches the expected values using assertions.

```java
public class TireTest {

    @Test
    public void testTireCreation() {
        ArrayList<List<String>> warehouse = new ArrayList<>();
        Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
          13), 7.5, true, 4, 100.0);
        assertEquals("195/65R15", tire.getSizeId());
        assertEquals("Petlas", tire.getBrand());
        assertEquals("car", tire.getCategory());
        assertEquals(50, tire.getSalesRate());
        assertEquals(LocalDate.of(2024, 11, 13), tire.getExpiringDate());
        assertEquals(LocalDate.of(2024, 10, 13), tire.getManufactureDate());
        assertEquals(7.5, tire.getTreadDepth());
        assertTrue(tire.isFromFactory());
        assertEquals(4, tire.getNumberOfTires());
        assertEquals(100.0, tire.getOriginalPrice());
    }
```

Test Explanation : This test ensures that the constructor of the Tire class correctly initializes all fields.

Scenario 2: testOrderRequirement

Objective: Verify that the number of tires can be updated and checked against a requirement.
Setup:

- A Tire object is created with 11 tires.
- The number of tires is updated to 10.

Execution and Assertion:

- The test checks if the updated number of tires is less than 12.

```java
    @Test
    public void testOrderRequirement() {
        ArrayList<List<String>> warehouse = new ArrayList<>();
        Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
          13), 7.5, true, 11, 100.0);
        tire.setNumberOfTires(10);
        assertTrue(tire.getNumberOfTires() < 12);
    }
```

This test ensures that the setNumberOfTires method works correctly and that the number of tires can be compared against a threshold.

Scenario 3 : testTireExpirationSoon

Objective: Check if the tire's expiring date is within a certain period (30 days from now).
Setup:

- A Tire object is created with an expiring date set to 20 days from now.

Execution and Assertion:

- The test checks if the expiring date is before 30 days from now.

```
35      @Test
36      public void testTireExpirationSoon() {
37          ArrayList<List<String>> warehouse = new ArrayList<>();
38          Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.now().plusDays(20), warehouse, LocalDate.of(2024, 10,
              13), 7.5, true, 4, 100.0);
39          assertTrue(tire.getExpiringDate().isBefore(LocalDate.now().plusDays(30)));
40      }
```

Test Explanation : This test ensures that the tire's expiring date can be correctly compared to a future date.

Scenario 4 : testTireTreadDepth

Setup:

- A Tire object is created with a tread depth of 6.5 mm.

Execution and Assertion:

- The test checks if the tread depth is less than 7 mm.

Objective: Validate that the tire's tread depth can be checked against a threshold.

```
42      @Test
43      public void testTireTreadDepth() {
44          ArrayList<List<String>> warehouse = new ArrayList<>();
45          Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
              13), 6.5, true, 4, 100.0);
46          assertTrue(tire.getTreadDepth() < 7);
47      }
```

Test Explanation : This test ensures that the tread depth can be correctly checked and compared against a value.

Scenario 5: testTireFromFactory

Objective : Verify that the tire's factory status can be updated and checked.
Setup:

- A Tire object is created with isFromFactory set to true.

Execution and Assertion:

- The test first checks if isFromFactory is true.
- The factory status is then updated to false, and the test checks if isFromFactory is false.

```java
@Test
public void testTireFromFactory() {
    ArrayList<List<String>> warehouse = new ArrayList<>();
    Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
      13), 7.5, true, 4, 100.0);
    assertTrue(tire.isFromFactory());
    tire.setFromFactory(false);
    assertFalse(tire.isFromFactory());
}
```

Test Explanation: This test ensures that the isFromFactory attribute can be updated and accurately reflects the tire's factory status.

Scenario 6: testTirePriceUpdate

Objective : Ensure that the tire's original and discounted prices can be updated and retrieved correctly.
Setup:

- A Tire object is created with an original price of 100.0.

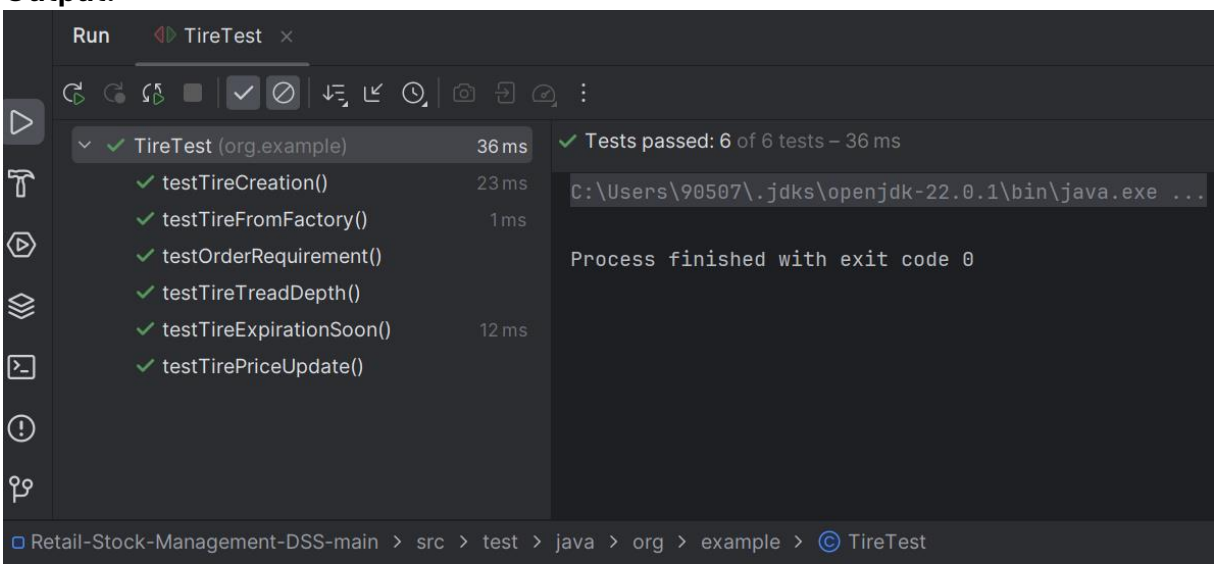
Execution and Assertion:

- The original price is updated to 120.0, and the test checks if the new original price is 120.0.
- The discounted price is set to 100.0, and the test checks if the discounted price is 100.0.

```java
@Test
public void testTirePriceUpdate() {
    ArrayList<List<String>> warehouse = new ArrayList<>();
    Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  warehouse, LocalDate.of(2024, 10,
      13), 7.5, true, 4, 100.0);
    tire.setOriginalPrice(120.0);
    assertEquals(120.0, tire.getOriginalPrice());
    tire.setDiscountedPrice(100.0);
    assertEquals(100.0, tire.getDiscountedPrice());
}
}
```

Test Explanation : This test ensures that the setOriginalPrice and setDiscountedPrice methods work correctly and that the prices are updated and retrieved accurately.

Output:



Summary of Tire Test:

These test cases comprehensively verify various functionalities of the Tire class, including object creation, attribute updates, and logical comparisons. Each test case isolates a specific feature or requirement, ensuring that the Tire class behaves as expected in different scenarios.

6.4 Warehouse Test

Scenario 1 : testAddTire

Objective: Ensure that a Tire object can be added to the Warehouse and that the Warehouse correctly tracks the number of tires.

Setup:

- A Warehouse object named "Test Warehouse" is created.
- A Tire object is created with specific attributes.

Execution:

- The Tire object is added to the Warehouse using the addTire method.

Assertion:

- The test verifies that the size of the tires list in the Warehouse is 1, indicating that one tire has been added.

```java
public class WarehouseTest {

    @Test
    public void testAddTire() {
        Warehouse warehouse = new Warehouse("Test Warehouse");
        ArrayList<List<String>> wh = new ArrayList<>();
        Tire tire = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  wh, LocalDate.of(2024, 10, 13), 7.5,
            true, 4, 100.0);
        warehouse.addTire(tire);
        assertEquals(1, warehouse.getTires().size());
    }
```

Test Explanation : This test ensures that the addTire method correctly adds a tire to the Warehouse and updates the internal list of tires.

Scenario 2 : testGetTires

Objective : Ensure that the Warehouse can store and retrieve multiple Tire objects.

Setup:

- A Warehouse object named "Test Warehouse" is created.
- Two Tire objects are created with different attributes.

Execution:

- Both Tire objects are added to the Warehouse using the addTire method.

Assertion:

- The test verifies that the size of the tires list in the Warehouse is 2, indicating that both tires have been added.
- This assertion is done twice for verification.

```java
    @Test
    public void testGetTires() {
        Warehouse warehouse = new Warehouse("Test Warehouse");
        ArrayList<List<String>> wh = new ArrayList<>();
        Tire tire1 = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  wh, LocalDate.of(2024, 10, 13),
          7.5, true, 4, 100.0);
        Tire tire2 = new Tire("225/45R17", "Bridgestone", "car", 60, LocalDate.of(2024, 12, 13),  wh, LocalDate.of(2024, 10,
          20), 8.0, false, 5, 120.0);
        warehouse.addTire(tire1);
        warehouse.addTire(tire2);

        assertEquals(2, warehouse.getTires().size());
        assertEquals(2, warehouse.getTires().size());
    }
```

Test Explanation: This test ensures that the Warehouse can correctly store and retrieve multiple tires and that the list of tires reflects the correct count.

Scenario 3: testGetAllTires

Objective: Ensure that the Warehouse can retrieve all tires grouped by their brand.
Setup:

- A Warehouse object named "Test Warehouse" is created.

- Two Tire objects from different brands are created.

Execution:

- Both Tire objects are added to the Warehouse using the addTire method.

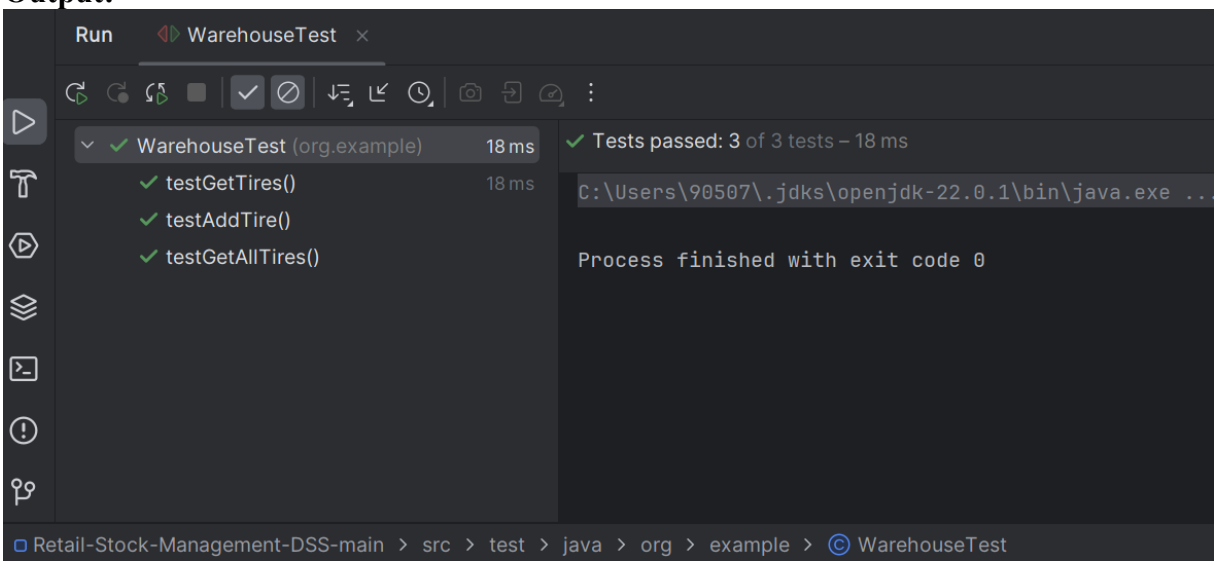
Assertion:

- The test verifies that the size of the all tires map in the Warehouse is 2, indicating that two different brands of tires are present.
- The test checks that the Warehouse contains keys for both "Petlas" and "Bridgestone" in the map returned by getAllTires.

```
33      @Test
34      public void testGetAllTires() {
35          Warehouse warehouse = new Warehouse("Test Warehouse");
36          ArrayList<List<String>> wh = new ArrayList<>();
37          Tire tire1 = new Tire("195/65R15", "Petlas", "car", 50, LocalDate.of(2024, 11, 13),  wh, LocalDate.of(2024, 10, 13),
              7.5, true, 4, 100.0);
38          Tire tire2 = new Tire("225/45R17", "Bridgestone", "car", 60, LocalDate.of(2024, 12, 13),  wh, LocalDate.of(2024, 10,
              20), 8.0, false, 5, 120.0);
39          warehouse.addTire(tire1);
40          warehouse.addTire(tire2);
41
42          assertEquals(2, warehouse.getAllTires().size());
43          assertTrue(warehouse.getAllTires().containsKey("Petlas"));
44          assertTrue(warehouse.getAllTires().containsKey("Bridgestone"));
45      }
46  }
```

Test Explanation: This test ensures that the Warehouse can correctly group and retrieve tires by their brand, and that the method getAllTires returns a map with the expected keys.

Output:



Summary of Warehouse Test:
These test cases comprehensively verify various functionalities of the Warehouse class, including the addition of tires, retrieval of tire lists, and grouping of tires by brand. Each test case isolates a specific feature or requirement, ensuring that the Warehouse class behaves as expected in different scenarios.

7 Conclusion and Evaluation

7.1 Problem

Effective tire inventory management poses challenges such as maintaining accurate stock levels, tracking expiration dates, and ensuring efficient inventory handling. Inadequate systems or reliance on manual processes can lead to stockouts, overstocking, and inventory mismanagement.

7.2 Key Results from the Project

The tire inventory management system demonstrated significant improvements in accuracy and efficiency. By implementing the solutions, stockouts were reduced, overstocking was minimized, and inventory management processes were streamlined. The system provided a clear overview of tire stock levels, expiration dates, and sales rates, enabling better decision-making.

7.3 Strengths and Weaknesses

Strengths:

- **Accuracy:** The detailed scoring algorithm effectively prioritized inventory based on multiple factors, resulting in more accurate stock management.
- **User-Friendliness:** The intuitive GUI made it easy for users to navigate and manage tire inventory, reducing the learning curve and increasing productivity.
- **Real-Time Tracking:** Real-time inventory tracking prevented stockouts and overstocking, ensuring that the right tires were always available.

Weaknesses:

- **Initial Setup Complexity:** Setting up the system initially required significant time and effort to configure and input existing data.
- **Dependency on Accurate Data:** The system's effectiveness depended heavily on the accuracy of the input data. Inaccurate data could lead to incorrect inventory decisions.

7.4 Recommendations and Future Work

To further enhance the tire inventory management system, the following recommendations are made:

- **Integration with Other Systems:** Integrate the inventory management system with other business systems, such as sales and procurement, to create a more cohesive and automated workflow.
- **Advanced Analytics:** Implement advanced analytics and machine learning algorithms to predict future inventory needs and trends, further improving stock management.
- **Scalability Improvements:** Ensure the system can scale to accommodate larger inventories and multiple warehouse locations.
- **User Training Programs:** Develop comprehensive training programs to ensure users are fully equipped to utilize the system's features effectively.

7.5 Solution

The project developed and implemented a tire inventory management system that included a detailed scoring algorithm, a user-friendly interface, real-time inventory tracking, and comprehensive reporting capabilities. These solutions addressed the key challenges and improved overall inventory management efficiency.

7.6 Real-World Data Usage

The solutions and improvements were tested and applied using real-world data, not simulated data. This ensured the results and benefits observed were practical and directly applicable to real-world tire inventory management scenarios.

7.7 To Conclude:

This project focuses on facilitating stock management in a systematic and organized manner for products stored in a mixed fashion within a 1000 m² warehouse, emphasizing logistics. It was developed with the purpose of providing a decision support system for efficient storage management.