Assignment 2

Name: Tianfu Xu

Course: CS-665 Software Designs & Patterns

Date: 09/29/2023

Shop.java

```
package edu.bu.met.cs665.example1;
import java.util.ArrayList;
import java.util.List;

public class Shop {
    private String name;
    private List<Driver> drivers = new ArrayList<>();

    public Shop(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addObserver(Driver driver) {
        drivers.add(driver);
    }

    public void removeObserver(Driver driver) {
        drivers.remove(driver);
    }

    public void generateDeliveryRequest() {
        DeliveryRequest request = new DeliveryRequest(this);
        notifyDrivers(request);
    }

    private void notifyDrivers(DeliveryRequest request) {
        for (Driver driver : drivers) {
            driver.receiveDeliveryRequest(request);
        }
    }
}
```

Explanation:

This class manages a list of drivers who can receive delivery requests from the shop. It maintains the shop's name as a private field and offers methods to interact with the list of drivers, such as adding and removing observers. The generateDeliveryRequest method creates a new delivery request associated with the shop, and subsequently notifies all registered drivers about the request. This implementation utilizes the Observer pattern to establish a communication mechanism between the shop and its drivers, allowing for efficient delivery request handling. The use of a list to manage drivers ensures that the shop can dynamically engage with multiple drivers as part of its delivery operations.

Driver.java

```
package edu.bu.met.cs665.example1;
public class Driver {
    private String name;

    public Driver(String name) {
        this.name = name;
    }

    public void receiveDeliveryRequest(DeliveryRequest request) {
        System.out.println(name + " received delivery request from " +
request.getShop().getName());
    }
}
```

Explanation:

The `Driver` class represents a delivery driver in the system. It has a private field `name` to store the driver's name, which is set through the constructor when a `Driver` object is created. The class contains a method `receiveDeliveryRequest` that takes a `DeliveryRequest` object as a parameter. This method is responsible for processing delivery requests received by the driver. It uses the `System.out.println` statement to print a message indicating that the driver, identified by their name, has received a delivery request from a specific shop, obtained through the `request.getShop().getName()` method call. This class encapsulates the behavior and attributes of a driver in the system, providing the necessary functionality for handling delivery requests.

DeliveryRequest.java

```
package edu.bu.met.cs665.example1;

public class DeliveryRequest {

    private Shop shop;


    public DeliveryRequest(Shop shop) {

        this.shop = shop;

    }


    public Shop getShop() {

        return shop;

    }

}
```

Explanation:

The `DeliveryRequest` class represents a specific delivery request made by a shop. It contains a private field `shop` which refers to the shop that generated this delivery request. The constructor takes a `Shop` object as a parameter, allowing the `DeliveryRequest` to be associated with a specific shop upon creation. The class provides a `getShop` method, which allows other parts of the program to retrieve the associated shop. This design ensures that each delivery request is linked to a specific shop, facilitating tracking and handling of the request throughout the system.

Main.java

```
package edu.bu.met.cs665;

import edu.bu.met.cs665.example1.Driver;
import edu.bu.met.cs665.example1.Shop;

public class Main {
  public static void main(String[] args) {

    Shop shop = new Shop("Amazon Store");

      Driver driver1 = new Driver("John Doe");
      Driver driver2 = new Driver("Jane Doe");
      Driver driver3 = new Driver("Bob Smith");
      Driver driver4 = new Driver("Alice Johnson");
      Driver driver5 = new Driver("Eve Davis");

      shop.addObserver(driver1);
      shop.addObserver(driver2);
      shop.addObserver(driver3);
      shop.addObserver(driver4);
      shop.addObserver(driver5);

      shop.generateDeliveryRequest();
  }
}
```

Explanation:
The Main class serves as the entry point for executing the program. In this class, a shop named "Amazon Store" is created, and five drivers with distinct names ("John Doe", "Jane Doe", "Bob Smith", "Alice Johnson", and "Eve Davis") are instantiated. Each driver is then registered as an observer of the shop, allowing them to receive delivery requests. Finally, a delivery request is generated, initiating the notification process to all registered drivers. This class orchestrates the interaction between the shop and its observers, demonstrating the functionality of the observer pattern in action.

- Explain the level of flexibility in your implementation, including how new object types can be easily added or removed in the future.

Flexibility in Implementation:

The implementation exhibits a high level of flexibility. The use of object-oriented principles like encapsulation and dependency injection allows for easy integration of new object types. For example, introducing a new entity like a Product or a Customer would involve creating a respective class and ensuring it adheres to established interfaces or extends existing classes. The Observer design pattern employed in the Shop class also enhances flexibility, as it allows for dynamic addition and removal of observers (drivers) without modifying existing code. This ensures that the system can evolve to accommodate new requirements and entities seamlessly.

- Discuss the simplicity and understandability of your implementation, ensuring that it is easy for others to read and maintain.

Simplicity and Understandability:

The code is designed with simplicity and understandability in mind. Each class is named descriptively and follows the Single Responsibility Principle, meaning that each class has a clear and distinct purpose. The methods have intuitive names, making it easy to comprehend their functionality. The relationships between classes are straightforward, promoting clarity in the code's structure. Additionally, comments have been used where necessary to provide further clarity on the code's functionality. This simplicity and clarity make the codebase accessible and maintainable for other developers.

- Describe how you have avoided duplicated code and why it is important.

Avoidance of Duplicated Code:

The implementation successfully avoids duplicated code. Each class and method serves a specific purpose and encapsulates functionality appropriately. For instance, the generateDeliveryRequest method in the Shop class encapsulates the logic for creating a new DeliveryRequest. This ensures that the logic for generating delivery requests is centralized, reducing redundancy and promoting maintainability. Moreover, the Observer pattern eliminates the need for explicit notification code in the Main class, as the Shop class handles the notification to all registered observers. This design choice prevents duplicated notification logic across multiple places

- If applicable, mention any design patterns you have used and explain why they were chosen.

Design Patterns:

The implementation incorporates the Observer design pattern, which allows for a loosely coupled relationship between the Shop and Driver classes. This pattern is chosen to facilitate communication between the shop and drivers without direct dependencies. By registering drivers as observers, the shop can notify all registered drivers about new delivery requests. This promotes scalability, as it enables the system to accommodate additional observers or entities in the future without necessitating modifications to existing code. The Observer pattern also enhances code maintainability by separating concerns related to notification from the core logic of the shop.

UML Class Diagram