FACULTY OF ENGINEERING AND APPLIED SCIENCE

Software Design and Architecture SOFE 3650U

Lab 1 Report

Course Instructor	Harshvardhan Singh
Due Date	September 17, 2024
Group number	Group 8
Group Members	Mohammed Nasser - 100852276 Mahnoor Jamal - 100822030 Yordanos Keflinkiel - 100867864

Questions:

The Adapter Pattern is a structural design pattern that converts one class's interface into another that clients expect, allowing incompatible interfaces to work together. Also known as the wrapper pattern, it enables collaboration between objects with different interfaces without altering their existing code. This pattern is useful for reusing classes with differing interfaces and prevents inflexibility caused by changing class interfaces. It involves a Client, Target, Adapter, and Adaptee, with the Adapter acting as a bridge. Although it adds a layer of abstraction, it enhances flexibility. Commonly used in systems like Java's I/O framework, it shares similarities with the Decorator and Bridge patterns.

The Observer Pattern is a behavioural design pattern that establishes a one-to-many relationship, where a change in the subject’s state triggers updates to all dependent observers automatically. Known also as Dependents or Publish-Subscribe, it is used to keep multiple objects synchronized with a single subject without tight coupling. For instance, in GUI frameworks, a button click (subject) notifies all registered listeners (observers). The pattern involves a Subject maintaining a list of Observers, with ConcreteSubjects notifying ConcreteObservers of state changes. It supports loose coupling and dynamic observer management but can face performance challenges with frequent updates. It’s widely used in GUIs and MVC architectures and is related to the Mediator pattern and Publish-Subscribe mechanism.

Motivation (Forces):

For the Adapter Pattern, imagine a garage door system designed to control standard doors that only offer basic open and close functions. Now, a premium door with variable speed control is introduced, but the system still expects the standard interface. The Adapter Pattern allows the premium door to integrate smoothly with the existing system by adapting its interface, without modifying the core functionality of either the door or the system.

In the case of the Observer Pattern, think of a store that needs to notify multiple customers when a discount becomes available. Without the Observer Pattern, the store would have to update each customer manually, creating tight coupling. By using the Observer Pattern, the store can automatically notify all registered customers when a discount changes, decoupling the store from individual customers and simplifying the notification process.

Applicability:

The Adapter Pattern is useful when you need to integrate new components with incompatible interfaces into existing systems. It's particularly effective for reusing existing classes or maintaining backward compatibility. For instance, if a system designed for standard doors needs to support premium doors with additional features, the Adapter allows this integration without altering the system's original interface.

The Observer Pattern is helpful when multiple objects need to respond to changes in the state of a single object. It is commonly used in situations such as user interfaces, stock price monitoring, and event-driven applications where changes in one component must be reflected across many others. The Observer Pattern ensures that all relevant objects are updated automatically, while decoupling the subject from its observers.

Structure:

In the Adapter Pattern, the client communicates with the adapter, which implements the interface the client expects and converts the request into a format the adaptee (the class with the incompatible interface) can understand. In the garage door scenario, the GarageDoorAdapter serves as the middleman, adapting the standard open/close interface to work with the premium door's speed-control feature.

In the Observer Pattern, the subject keeps track of a list of observers. When its state changes, it informs all observers by calling their update() method, allowing them to react accordingly, such as adjusting their internal state. In the store example, the store acts as the subject and the customers are the observers. When the store's discount changes, it notifies all registered customers.

Participants:

The Adapter Pattern consists of three key components: the Client (the system expecting a standard interface), the Adapter (which translates requests to the new interface), and the Adaptee (the component that requires adaptation, such as the premium garage door).

In the Observer Pattern, the Subject (the store) keeps a list of Observers (customers). The Concrete Observers (individual customer instances) implement the update() method to process notifications from the subject.

Collaborations:
In the Adapter Pattern, the client sends a request to the adapter, which translates that request and delegates it to the adaptee, allowing both interfaces to work together without modification. For example, a client calls open() on a GarageDoorAdapter, and the adapter translates this into open(speed) for the premium garage door.

In the Observer Pattern, the subject notifies all registered observers when its state changes. The observers, in turn, update themselves based on the new information. For instance, when the store changes its discount, it notifies all customers, who then adjust their internal discount values accordingly.

Consequences:

The Adapter Pattern enables previously incompatible interfaces to work together without modifying the original components. This enhances code reuse and flexibility but may introduce additional complexity due to the extra layer of abstraction. It can also slightly increase overhead because of the need for interface translation.

The Observer Pattern decouples the subject from its observers, leading to a more maintainable and organized system. It allows for dynamic relationships between objects and simplifies adding or removing observers. However, if the subject has many observers or sends frequent notifications, it can lead to performance issues and potential inefficiencies.

Implementation:

When implementing the Adapter Pattern, the focus should be on adapting the interface rather than altering the underlying functionality. This can be done through delegation. In some programming languages, where multiple inheritance is not supported, delegation may be preferable to inheritance.

For the Observer Pattern, it’s crucial to handle the registration and unregistration of observers carefully to prevent problems such as memory leaks or dangling references. In languages with garbage collection, using weak references can help manage these issues effectively.

Sample Code:
In the Adapter Pattern, the adapter converts client requests such as open() into a format like open(speed) for the premium door. In the Observer Pattern, the subject invokes the update() method on all registered observers whenever its state changes.

Known Uses:

The Adapter Pattern is frequently employed in GUI toolkits to integrate new components into existing systems. It is also used in libraries to ensure backward compatibility with older versions of an API.

The Observer Pattern is widely utilized in user interface frameworks (such as event listeners), stock monitoring systems, and other event-driven architectures where updates must be communicated to multiple subscribers.

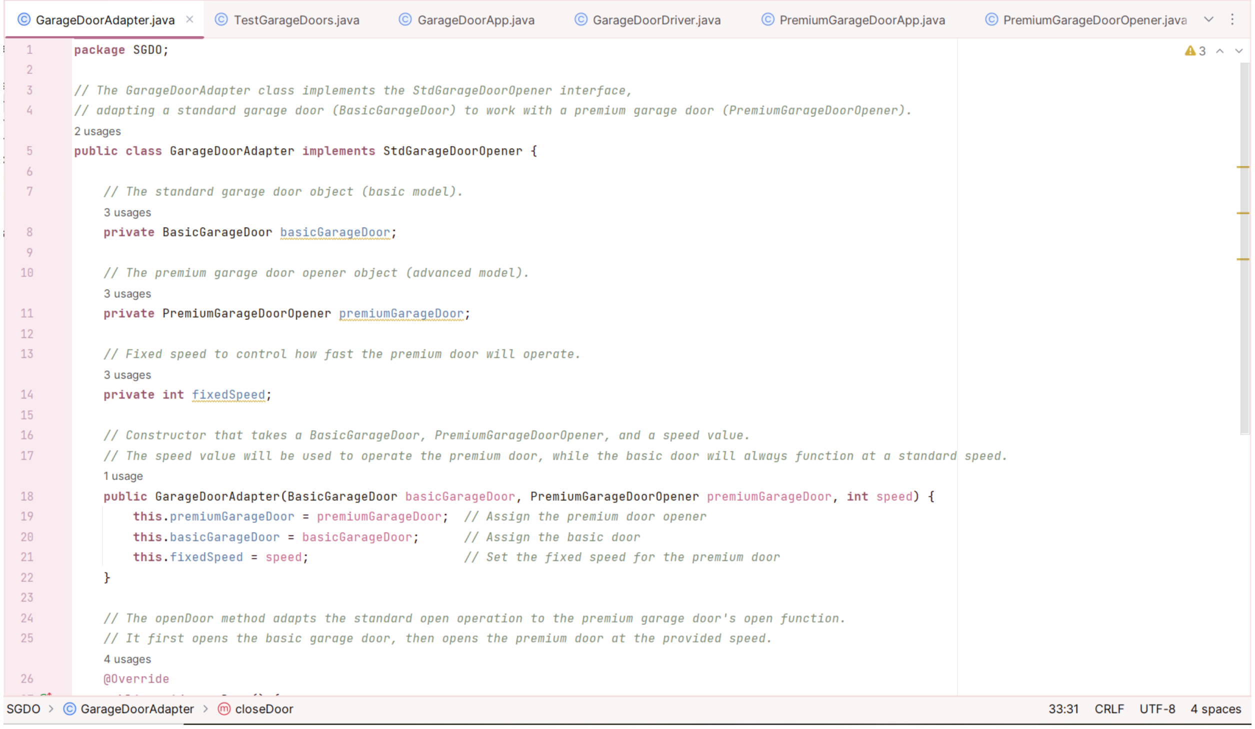
Related Patterns:

The Adapter Pattern is similar to the Bridge Pattern, which also separates abstraction from implementation but places more emphasis on decoupling the two. Additionally, it relates to the Decorator Pattern, which modifies an object's behaviour, while the Adapter Pattern focuses solely on altering the interface.

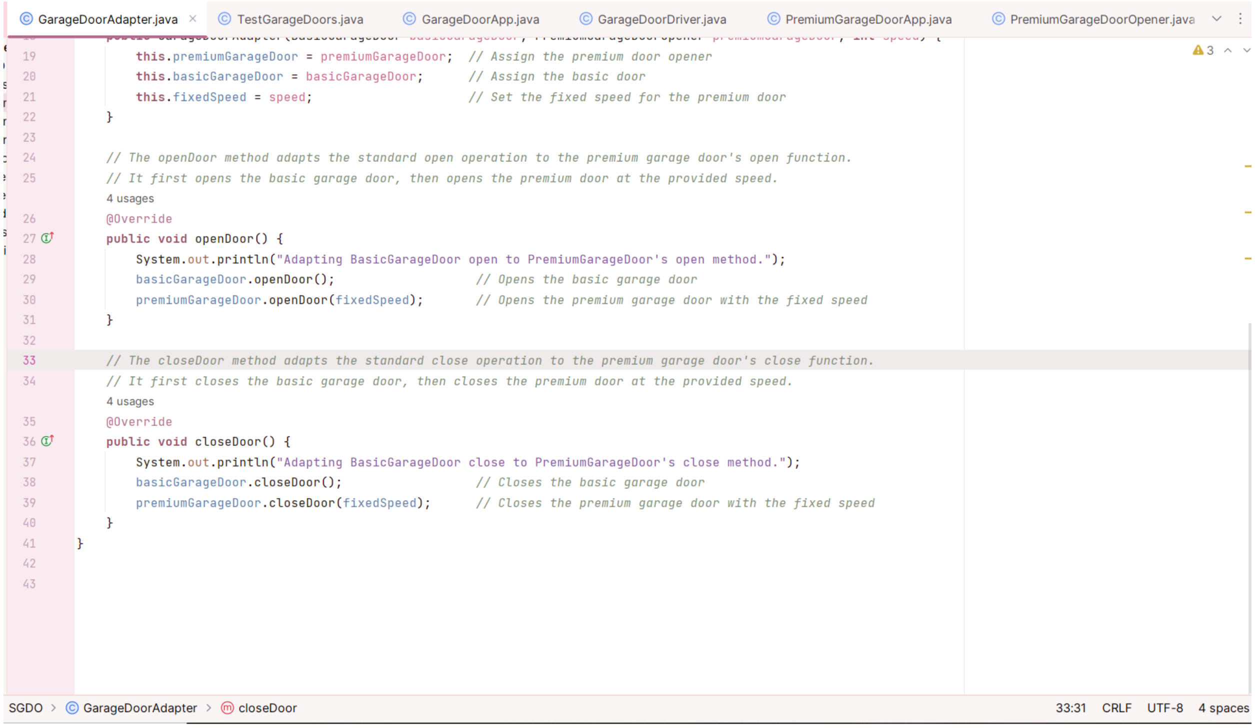
The Observer Pattern is similar to the Mediator Pattern, which minimizes direct interactions between objects. It also bears resemblance to Event-Driven Architecture, where events trigger responses from various components, similar to how observers react to changes in the subject.

Code Screenshots and Test Run

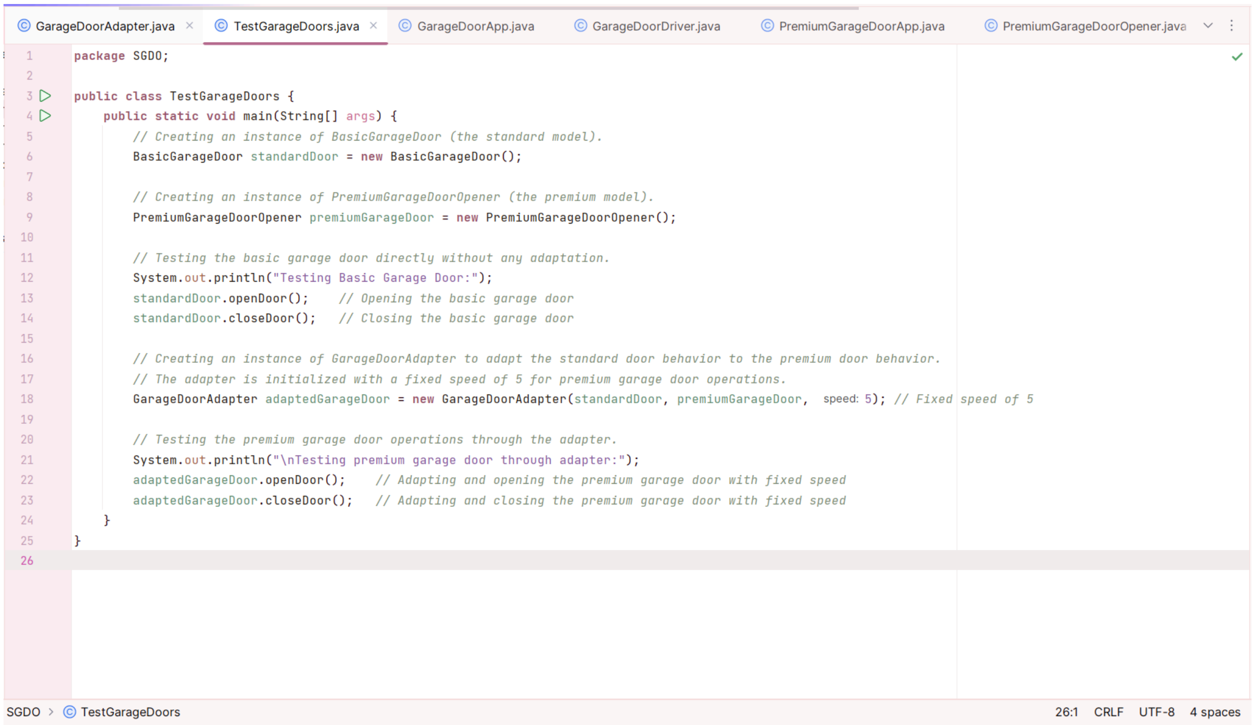
Adapter Pattern



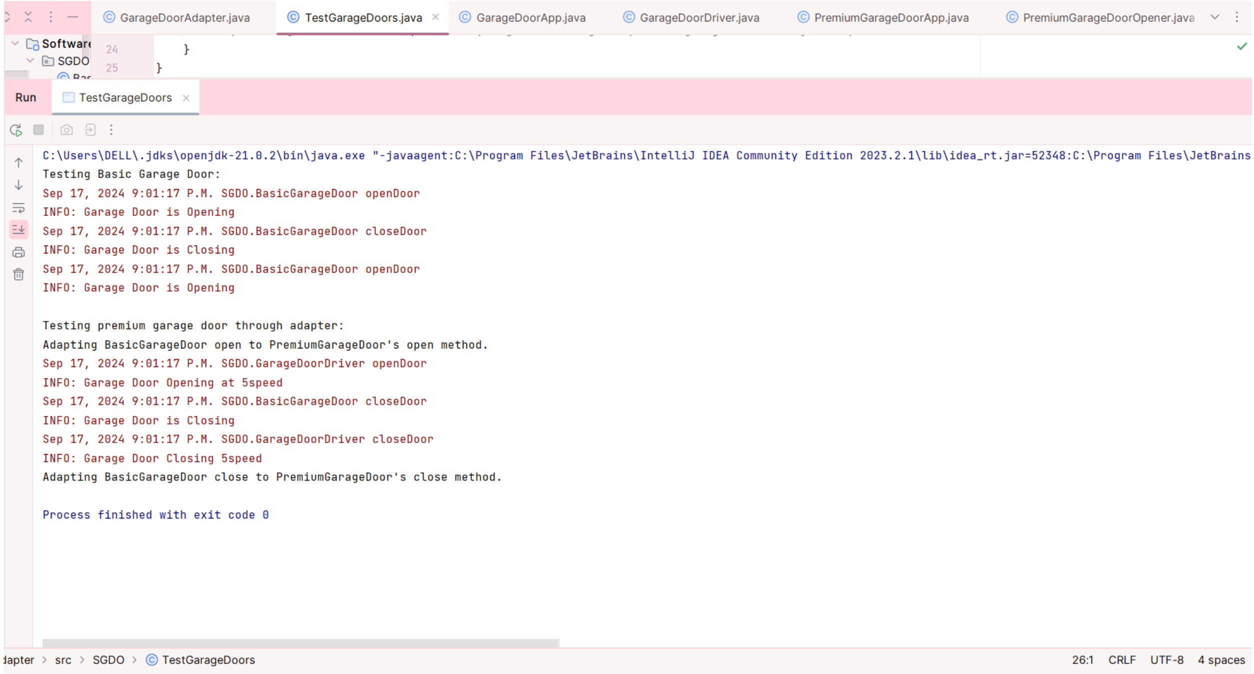
```java
package SGDO;

// The GarageDoorAdapter class implements the StdGarageDoorOpener interface,
// adapting a standard garage door (BasicGarageDoor) to work with a premium garage door (PremiumGarageDoorOpener).
public class GarageDoorAdapter implements StdGarageDoorOpener {

    // The standard garage door object (basic model).
    private BasicGarageDoor basicGarageDoor;

    // The premium garage door opener object (advanced model).
    private PremiumGarageDoorOpener premiumGarageDoor;

    // Fixed speed to control how fast the premium door will operate.
    private int fixedSpeed;

    // Constructor that takes a BasicGarageDoor, PremiumGarageDoorOpener, and a speed value.
    // The speed value will be used to operate the premium door, while the basic door will always function at a standard speed.
    public GarageDoorAdapter(BasicGarageDoor basicGarageDoor, PremiumGarageDoorOpener premiumGarageDoor, int speed) {
        this.premiumGarageDoor = premiumGarageDoor;  // Assign the premium door opener
        this.basicGarageDoor = basicGarageDoor;      // Assign the basic door
        this.fixedSpeed = speed;                     // Set the fixed speed for the premium door
    }

    // The openDoor method adapts the standard open operation to the premium garage door's open function.
    // It first opens the basic garage door, then opens the premium door at the provided speed.
    @Override
```


GarageDoorAdapter.java
TestGarageDoors.java
GarageDoorApp.java
GarageDoorDriver.java
PremiumGarageDoorApp.java
PremiumGarageDoorOpener.java
⚠3
4 usages
4 usages
SGDO › GarageDoorAdapter › closeDoor
33:31 CRLF UTF-8 4 spaces

```java
        this.premiumGarageDoor = premiumGarageDoor;  // Assign the premium door opener
        this.basicGarageDoor = basicGarageDoor;      // Assign the basic door
        this.fixedSpeed = speed;                     // Set the fixed speed for the premium door
    }

    // The openDoor method adapts the standard open operation to the premium garage door's open function.
    // It first opens the basic garage door, then opens the premium door at the provided speed.
    @Override
    public void openDoor() {
        System.out.println("Adapting BasicGarageDoor open to PremiumGarageDoor's open method.");
        basicGarageDoor.openDoor();                    // Opens the basic garage door
        premiumGarageDoor.openDoor(fixedSpeed);        // Opens the premium garage door with the fixed speed
    }

    // The closeDoor method adapts the standard close operation to the premium garage door's close function.
    // It first closes the basic garage door, then closes the premium door at the provided speed.
    @Override
    public void closeDoor() {
        System.out.println("Adapting BasicGarageDoor close to PremiumGarageDoor's close method.");
        basicGarageDoor.closeDoor();                   // Closes the basic garage door
        premiumGarageDoor.closeDoor(fixedSpeed);       // Closes the premium garage door with the fixed speed
    }
}
```

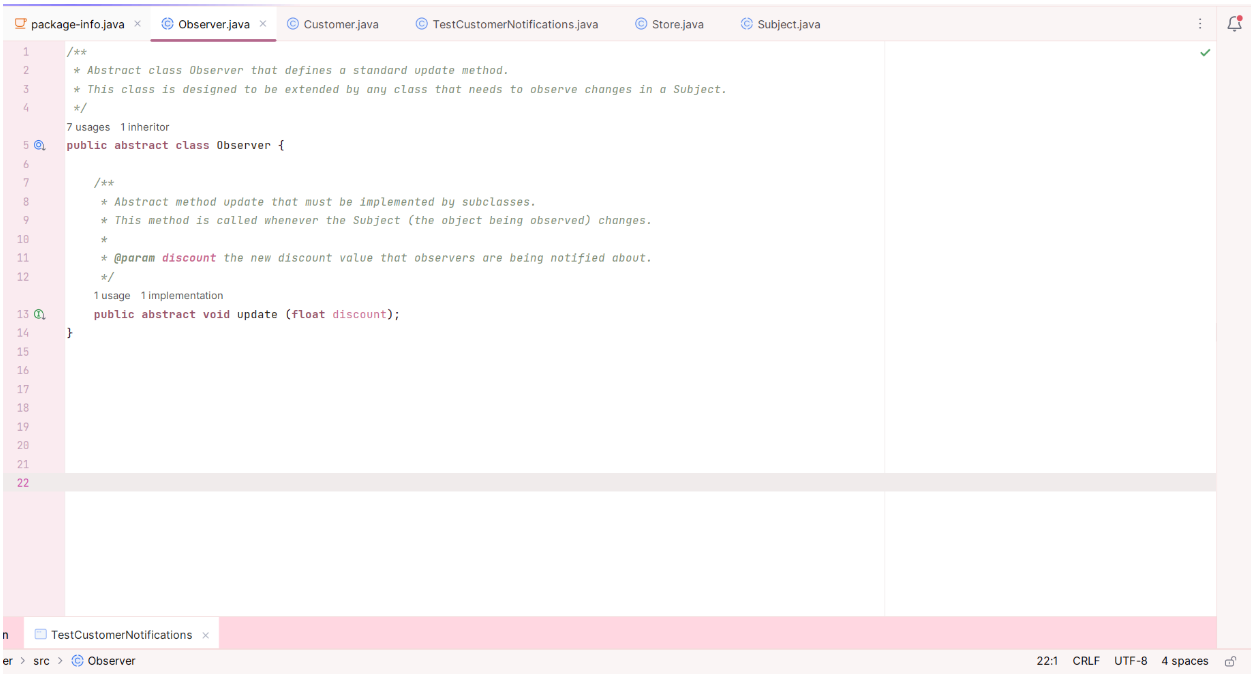


```java
package SGDO;

public class TestGarageDoors {
    public static void main(String[] args) {
        // Creating an instance of BasicGarageDoor (the standard model).
        BasicGarageDoor standardDoor = new BasicGarageDoor();

        // Creating an instance of PremiumGarageDoorOpener (the premium model).
        PremiumGarageDoorOpener premiumGarageDoor = new PremiumGarageDoorOpener();

        // Testing the basic garage door directly without any adaptation.
        System.out.println("Testing Basic Garage Door:");
        standardDoor.openDoor();    // Opening the basic garage door
        standardDoor.closeDoor();   // Closing the basic garage door

        // Creating an instance of GarageDoorAdapter to adapt the standard door behavior to the premium door behavior.
        // The adapter is initialized with a fixed speed of 5 for premium garage door operations.
        GarageDoorAdapter adaptedGarageDoor = new GarageDoorAdapter(standardDoor, premiumGarageDoor,  speed: 5); // Fixed speed of 5

        // Testing the premium garage door operations through the adapter.
        System.out.println("\nTesting premium garage door through adapter:");
        adaptedGarageDoor.openDoor();    // Adapting and opening the premium garage door with fixed speed
        adaptedGarageDoor.closeDoor();   // Adapting and closing the premium garage door with fixed speed
    }
}
```

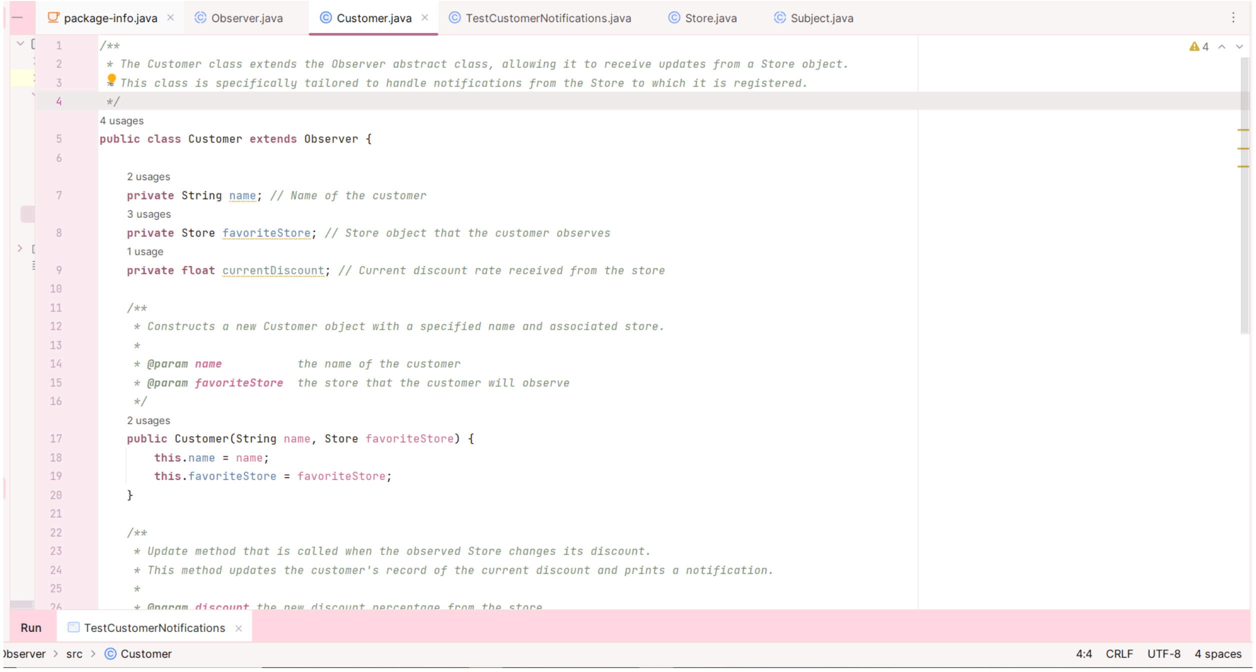


```
C:\Users\DELL\.jdks\openjdk-21.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.2.1\lib\idea_rt.jar=52348:C:\Program Files\JetBrains
Testing Basic Garage Door:
Sep 17, 2024 9:01:17 P.M. SGDO.BasicGarageDoor openDoor
INFO: Garage Door is Opening
Sep 17, 2024 9:01:17 P.M. SGDO.BasicGarageDoor closeDoor
INFO: Garage Door is Closing
Sep 17, 2024 9:01:17 P.M. SGDO.BasicGarageDoor openDoor
INFO: Garage Door is Opening

Testing premium garage door through adapter:
Adapting BasicGarageDoor open to PremiumGarageDoor's open method.
Sep 17, 2024 9:01:17 P.M. SGDO.GarageDoorDriver openDoor
INFO: Garage Door Opening at 5speed
Sep 17, 2024 9:01:17 P.M. SGDO.BasicGarageDoor closeDoor
INFO: Garage Door is Closing
Sep 17, 2024 9:01:17 P.M. SGDO.GarageDoorDriver closeDoor
INFO: Garage Door Closing 5speed
Adapting BasicGarageDoor close to PremiumGarageDoor's close method.

Process finished with exit code 0
```

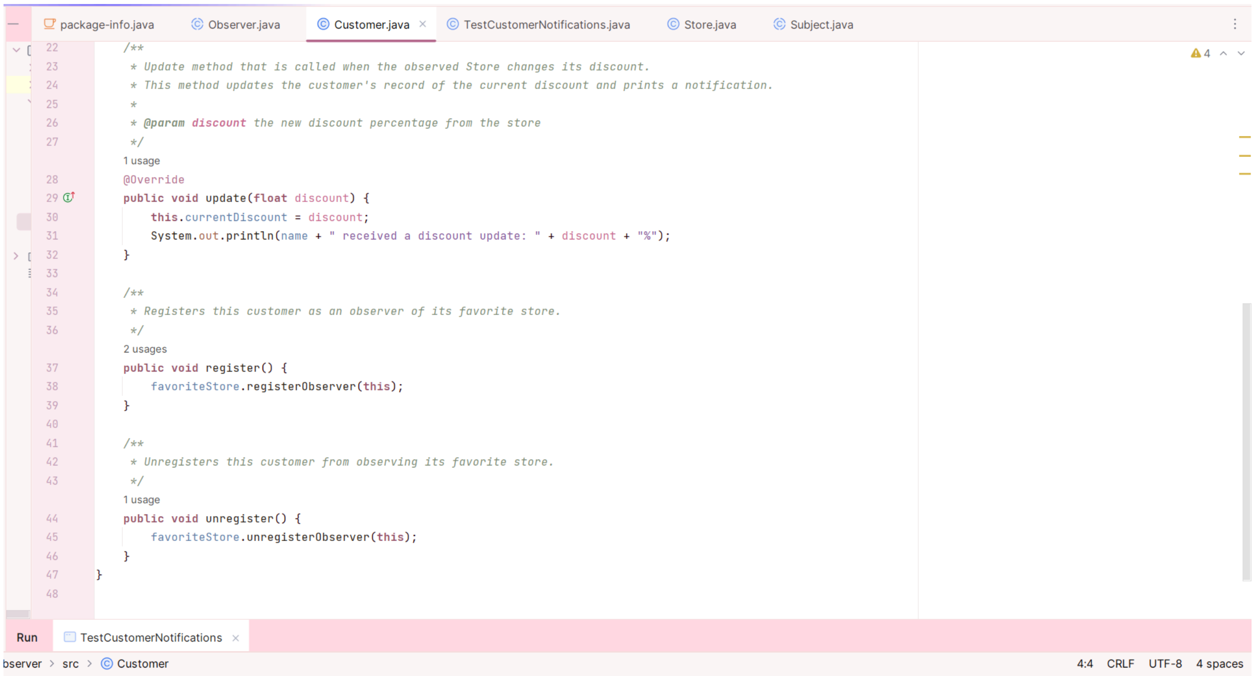
Observer Pattern



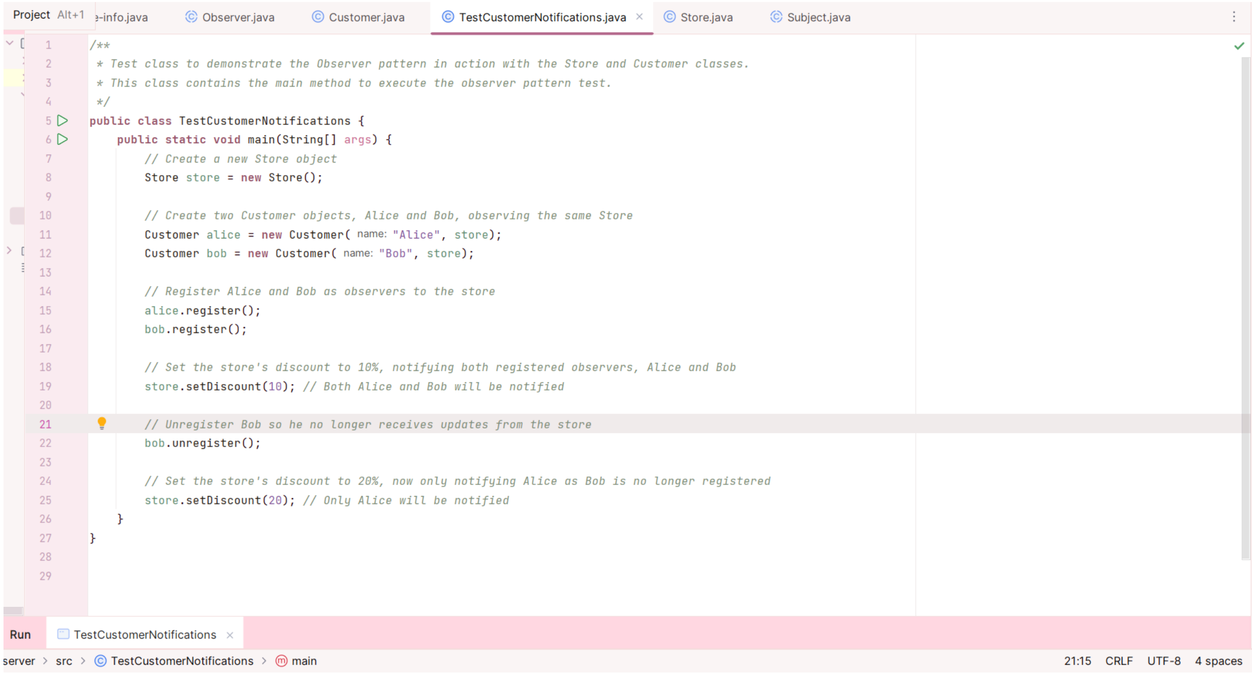
```java
/**
 * Abstract class Observer that defines a standard update method.
 * This class is designed to be extended by any class that needs to observe changes in a Subject.
 */
public abstract class Observer {

    /**
     * Abstract method update that must be implemented by subclasses.
     * This method is called whenever the Subject (the object being observed) changes.
     *
     * @param discount the new discount value that observers are being notified about.
     */
    public abstract void update (float discount);
}
```



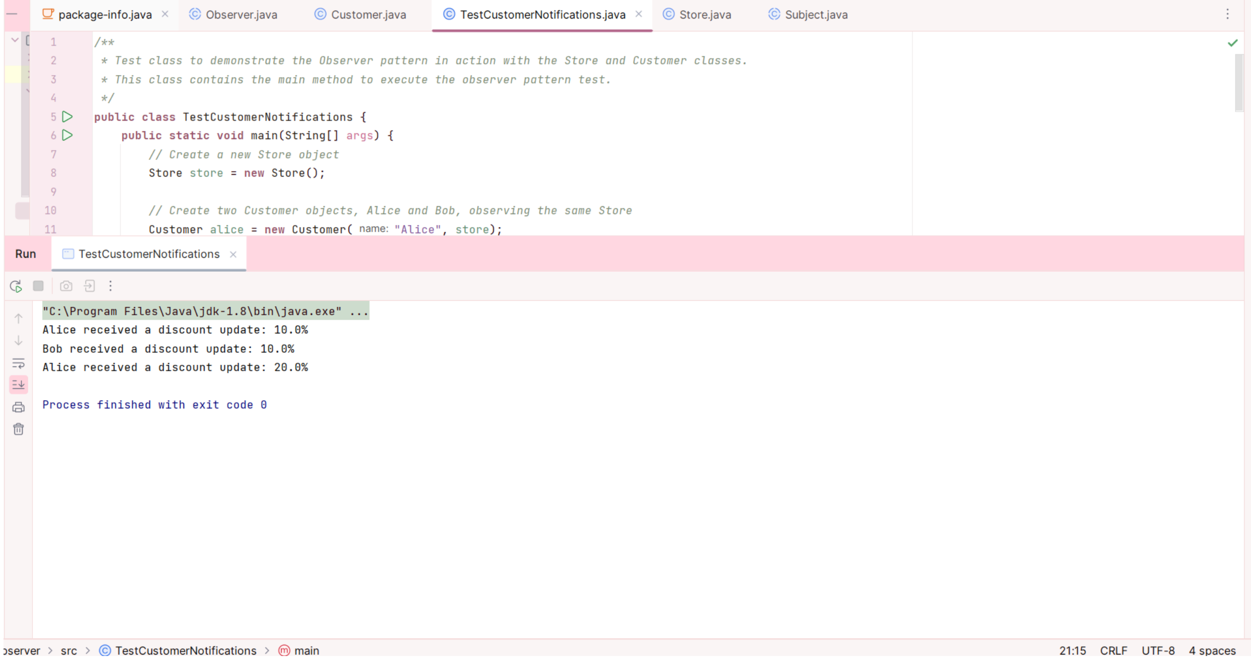
```java
/**
 * The Customer class extends the Observer abstract class, allowing it to receive updates from a Store object.
 * This class is specifically tailored to handle notifications from the Store to which it is registered.
 */
public class Customer extends Observer {

    private String name; // Name of the customer
    private Store favoriteStore; // Store object that the customer observes
    private float currentDiscount; // Current discount rate received from the store

    /**
     * Constructs a new Customer object with a specified name and associated store.
     *
     * @param name          the name of the customer
     * @param favoriteStore  the store that the customer will observe
     */
    public Customer(String name, Store favoriteStore) {
        this.name = name;
        this.favoriteStore = favoriteStore;
    }

    /**
     * Update method that is called when the observed Store changes its discount.
     * This method updates the customer's record of the current discount and prints a notification.
     *
     * @param discount the new discount percentage from the store
```



```java
    /**
     * Update method that is called when the observed Store changes its discount.
     * This method updates the customer's record of the current discount and prints a notification.
     *
     * @param discount the new discount percentage from the store
     */
    @Override
    public void update(float discount) {
        this.currentDiscount = discount;
        System.out.println(name + " received a discount update: " + discount + "%");
    }

    /**
     * Registers this customer as an observer of its favorite store.
     */
    public void register() {
        favoriteStore.registerObserver(this);
    }

    /**
     * Unregisters this customer from observing its favorite store.
     */
    public void unregister() {
        favoriteStore.unregisterObserver(this);
    }
}
```



```java
/**
 * Test class to demonstrate the Observer pattern in action with the Store and Customer classes.
 * This class contains the main method to execute the observer pattern test.
 */
public class TestCustomerNotifications {
    public static void main(String[] args) {
        // Create a new Store object
        Store store = new Store();

        // Create two Customer objects, Alice and Bob, observing the same Store
        Customer alice = new Customer( name: "Alice", store);
        Customer bob = new Customer( name: "Bob", store);

        // Register Alice and Bob as observers to the store
        alice.register();
        bob.register();

        // Set the store's discount to 10%, notifying both registered observers, Alice and Bob
        store.setDiscount(10); // Both Alice and Bob will be notified

        // Unregister Bob so he no longer receives updates from the store
        bob.unregister();

        // Set the store's discount to 20%, now only notifying Alice as Bob is no longer registered
        store.setDiscount(20); // Only Alice will be notified
    }
}
```


package-info.java
Observer.java
Customer.java
TestCustomerNotifications.java
Store.java
Subject.java
Run
TestCustomerNotifications
›server › src › TestCustomerNotifications › main
21:15
CRLF
UTF-8
4 spaces

```java
/**
 * Test class to demonstrate the Observer pattern in action with the Store and Customer classes.
 * This class contains the main method to execute the observer pattern test.
 */
public class TestCustomerNotifications {
    public static void main(String[] args) {
        // Create a new Store object
        Store store = new Store();

        // Create two Customer objects, Alice and Bob, observing the same Store
        Customer alice = new Customer( name: "Alice", store);
```

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Alice received a discount update: 10.0%
Bob received a discount update: 10.0%
Alice received a discount update: 20.0%

Process finished with exit code 0
```