**Exercise 7: Implementing the Observer Pattern**

**Define Subject Interface**

**Stock.java**

**package** mypackage;

**public** **interface** Stock {

**void** registerObserver(Observer observer);

**void** deregisterObserver(Observer observer);

**void** notifyObservers();

**void** setPrice(**double** price);

}

**Implement Concrete Subject**

**StockMarket.java**

**package** mypackage;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StockMarket **implements** Stock {

**private** List<Observer> observers = **new** ArrayList<>();

**private** **double** price;

**public** **void** registerObserver(Observer observer) {

observers.add(observer);

}

**public** **void** deregisterObserver(Observer observer) {

observers.remove(observer);

}

**public** **void** notifyObservers() {

**for** (Observer observer : observers) {

observer.update(price);

}

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

System.***out***.println("Stock price updated to: " + price);

notifyObservers();

}

}

**Define Observer Interface**

**Observer.java**

**package** mypackage;

**public** **interface** Observer {

**void** update(**double** price);

}

**Implement Concrete Observers**

**package** mypackage;

**public** **class** MobileApp **implements** Observer {

**private** String appName;

**public** MobileApp(String appName) {

**this**.appName = appName;

}

**public** **void** update(**double** price) {

System.***out***.println(appName + " (Mobile App) received stock price update: " + price);

}

}

**WebApp.java**

**package** mypackage;

**public** **class** WebApp **implements** Observer {

**private** String appName;

**public** WebApp(String appName) {

**this**.appName = appName;

}

**public** **void** update(**double** price) {

System.***out***.println(appName + " (Web App) received stock price update: " + price);

}

}

**Test the Observer Implementation**

**TestObserverPattern.java**

**package** mypackage;

**public** **class** TestObserverPattern {

**public** **static** **void** main(String[] args) {

StockMarket stockMarket = **new** StockMarket();

Observer mobileObserver = **new** MobileApp("StockMonitor");

Observer webObserver = **new** WebApp("StockWebPortal");

stockMarket.registerObserver(mobileObserver);

stockMarket.registerObserver(webObserver);

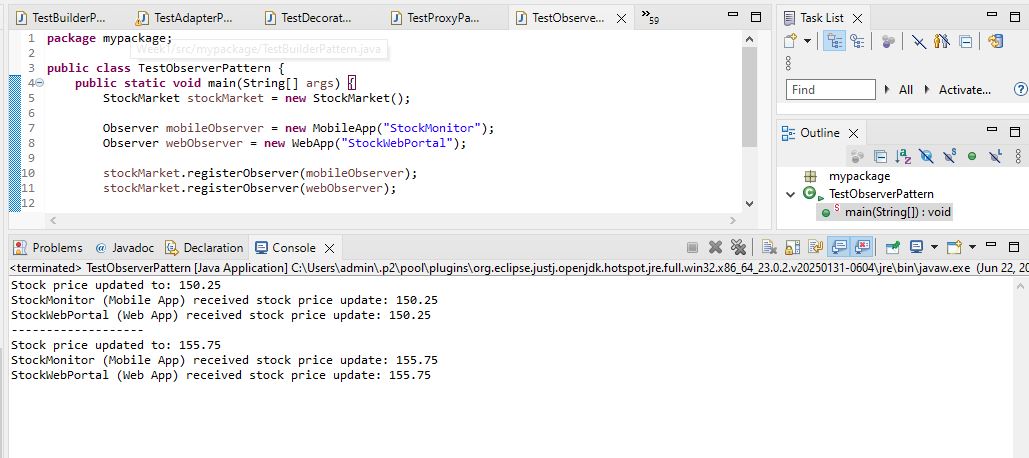
stockMarket.setPrice(150.25);

System.***out***.println("-------------------");

stockMarket.setPrice(155.75);

}

}

****

We are developing a stock market monitoring application where multiple clients need to receive stock price updates automatically. The Observer Pattern allows efficient notification to all registered observers when the stock price changes.

**1. Subject Interface**

The Stock interface defines methods for:

* registerObserver(): to add observers.
* deregisterObserver(): to remove observers.
* notifyObservers(): to notify all observers.
* setPrice(): to update the stock price.

**2. Concrete Subject**

The StockMarket class implements Stock and:

* Maintains a list of registered observers.
* Calls notifyObservers() whenever stock price changes.

**3. Observer Interface**

The Observer interface declares the method update() which is called by the subject.

**4. Concrete Observers**

* MobileApp and WebApp implement Observer.
* Each displays stock price updates when notified.

**5. Advantages of Observer Pattern**

* Allows dynamic addition/removal of observers.
* Promotes loose coupling between subject and observers.
* Automatic update of multiple components on state change.
* Scales easily as the number of observers grows.

**6. Time Complexity**

* Register/deregister: O(1)
* Notify observers: O(n), where n is the number of observers.

**7. Real-life Applications**

* Stock market monitoring systems.
* Event handling systems.
* GUI frameworks (event listeners).
* Social media notifications.