Karen Giselle Valdez Muñoz

Observer Pattern

Introduction

The Observer Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. In this pattern, a subject (or observable) maintains a list of observers (or listeners) that are dependent on the subject's state. When the subject's state changes, all registered observers are automatically notified and updated. In this document, we will explore the implementation of the Observer pattern in Java to solve the business case of stock notifications. This pattern helps in notifying various departments automatically whenever there is a change in the stock of a product.

Observer Interface

The Observer interface defines a contract for all observer classes. Each observer must implement the update method to receive notifications of changes.

```java

public interface Observer {
    void update(String product, int newStock);
}

```

- Method: update(String product, int newStock)
- Parameters:
    - product: The name of the product that has undergone a stock change.
    - newStock: The updated stock level of the product.

Subject Class (Product)

The Product class represents the subject in the Observer pattern. It maintains a list of observers and notifies them when its state changes.

```java
import java.util.ArrayList;


public class Product {
    private String name;
    private int stock;
    private List<Observer> observers = new ArrayList<>();

    public Product(String name, int stock) {
        this.name = name;
        this.stock = stock;
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setStock(int stock) {
        this.stock = stock;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name, stock);
        }
    }
}
```

- Fields:
    - name: The name of the product.
    - stock: The current stock level of the product.
    - observers: A list to hold all observers interested in stock updates.
- Constructor:
    - Initializes the name and stock of the product.
- Methods:
    - addObserver(Observer observer): Adds an observer to the list.
    - removeObserver(Observer observer): Removes an observer from the list.
    - setStock(int stock): Updates the stock level and triggers notifications.
    - notifyObservers(): Notifies all registered observers of the stock change.

Concrete Observer Classes

These classes implement the Observer interface and define specific behaviors upon receiving a stock update.

```
1
2  public class SalesDepartment implements Observer {
3
4      @Override
5      public void update(String product, int newStock) {
6          System.out.println("Sales notification. Product: " + product + ", New stock: " + newStock);
7      }
8
9  }
```

```
1
2  public class WarehouseDepartment implements Observer {
3
4      @Override
5      public void update(String product, int newStock) {
6          System.out.println("Warehouse Notification. Product: " + product + ", New stock: " + newStock
7      }
8
9  }
```

```
1
2  public class PurchaseDepartment implements Observer {
3
4      @Override
5      public void update(String product, int newStock) {
6          System.out.println("Purchases notification. Product: " + product + ", New stock: " + newStock
7      }
8
```

Method: update(String product, int newStock)

- Each concrete observer implements this method to handle stock updates in a department-specific way.
- The message printed reflects the specific department that has been notified.

Karen Giselle Valdez Muñoz

Main Class

The Main class demonstrates how to create a product, add observers, and simulate stock changes.

```java
public class principal {

    public static void main(String[] args) {
         // Create a product
        Product product = new Product("Laptop", 50);

        // Create observers
        Observer salesDept = new SalesDepartment();
        Observer purchaseDept = new PurchaseDepartment();
        Observer warehouseDept = new WarehouseDepartment();

        // Add observers to the product
        product.addObserver(salesDept);
        product.addObserver(purchaseDept);
        product.addObserver(warehouseDept);

        // Change stock and notify observers
        product.setStock(45); // Decrease stock
        product.setStock(30); // Decrease stock

    }

}
```

- Product Creation:
  - A Product object is instantiated with an initial stock of 50 units.
- Observer Creation:
  - Instances of SalesDepartment, PurchaseDepartment, and WarehouseDepartment are created.
- Adding Observers:
  - Each observer is registered with the product to receive notifications.
- Simulating Stock Changes:
  - The stock is updated twice, and each change triggers notifications to all registered observers.

Output:

```
Sales notification. Product: Laptop, New stock: 45
Purchases notification. Product: Laptop, New stock: 45
Warehouse Notification. Product: Laptop, New stock: 45
Sales notification. Product: Laptop, New stock: 30
Purchases notification. Product: Laptop, New stock: 30
Warehouse Notification. Product: Laptop, New stock: 30
```

Conclusion

The Observer pattern effectively decouples the subject (product) from its observers (departments). It allows multiple departments to receive real-time updates about stock changes without the subject needing to be aware of the specific observers. This design promotes flexibility and maintainability in the system.