Observer Pattern

Scenario:

Imagine you're developing a document editor that has various components, such as a text editor, spell checker, and formatting toolbar.

When the user makes changes to the document (like typing new text, changing styles, or applying formatting), you want to notify all

relevant components so they can update themselves accordingly. For example, the spell checker should highlight any new spelling errors,

and the formatting toolbar should reflect the current style.

Purpose:

The Observer Pattern defines a one-to-many dependency between objects, so that when one object (the subject) changes state, all its

dependents (observers) are notified and updated automatically. This pattern promotes loose coupling between the subject and its

observers, allowing for dynamic relationships between objects.

When to Use:

- When an object's state changes, and you want to automatically notify and update a set of dependent objects.
- When you want to establish a one-to-many relationship between objects, where one object can trigger updates in many other objects.
- When you want to decouple the subject from its observers, allowing either to change independently of the other.

Key Concepts:

- Subject: The object that holds the state and notifies observers when its state changes. It maintains a list of its observers.
- Observer: An interface or abstract class that defines the method to be called when the subject's state changes.
- Concrete Observer: Implements the Observer interface and defines how it should respond to notifications from the subject.
- Concrete Subject: Implements the Subject interface and maintains the state that the observers are interested in.

How It Works:

- Subject Interface: Defines methods for attaching, detaching, and notifying observers.
- Concrete Subject: Implements the Subject interface and maintains the state. It notifies observers when the state changes.
- Observer Interface: Defines the update() method that will be called when the subject's state changes.
- Concrete Observer: Implements the Observer interface and updates its state based on the notification from the subject.
- Client: Attaches observers to the subject and triggers state changes in the subject.

Implementation Example:

Here's how the implementation might look in Java:

```java
import java.util.ArrayList;
import java.util.List;

// Subject Interface
```

```
}

interface DocumentSubject {
    void attach(DocumentObserver observer);
    void detach(DocumentObserver observer);
    void notifyObservers();
}

// Concrete Subject
class TextDocument implements DocumentSubject {
    private List<DocumentObserver> observers = new ArrayList<>();
    private String content;

    public void setContent(String content) {
        this.content = content;
        notifyObservers();
    }

    public String getContent() {
        return content;
    }

    @Override
    public void attach(DocumentObserver observer) {
        observers.add(observer);
    }

    @Override
    public void detach(DocumentObserver observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (DocumentObserver observer : observers) {
            observer.update();
        }
    }
}

// Observer Interface
interface DocumentObserver {
    void update();
}

// Concrete Observer - Spell Checker
class SpellChecker implements DocumentObserver {
    private TextDocument document;

    public SpellChecker(TextDocument document) {
        this.document = document;
    }

    @Override
    public void update() {
                System.out.println("SpellChecker:   Checking   spelling   for   document   content:   "   +
document.getContent());
    }
}

// Concrete Observer - Formatting Toolbar
class FormattingToolbar implements DocumentObserver {
    private TextDocument document;

    public FormattingToolbar(TextDocument document) {
        this.document = document;
    }

    @Override
    public void update() {
            System.out.println("FormattingToolbar: Updating toolbar based on document content: " +
document.getContent());
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        TextDocument document = new TextDocument();

        SpellChecker spellChecker = new SpellChecker(document);
        FormattingToolbar toolbar = new FormattingToolbar(document);
        document.attach(spellChecker);
        document.attach(toolbar);

        System.out.println("Setting document content to 'Hello World!'");
        document.setContent("Hello World!");

        System.out.println("\nSetting document content to 'Observer Pattern in Java'");
        document.setContent("Observer Pattern in Java");
    }
}
```

Key Points to Remember:

- Observer Pattern allows an object (the subject) to notify multiple other objects (observers) about changes in its state. This pattern promotes loose coupling between the subject and observers, making it easier to modify or extend the system.
- One-to-Many Relationship: The pattern is useful for scenarios where one object needs to notify many others about changes, ensuring that the system remains consistent and up-to-date.

Advantages:

- Loose Coupling: The pattern decouples the subject from its observers, allowing them to vary independently.
- Dynamic Relationships: Observers can be added or removed at runtime, making the system flexible and adaptable to changes.
- Consistency: All observers are automatically notified and updated when the subject's state changes, ensuring consistency across the system.

Disadvantages:

- Potential for Cascading Updates: If not managed carefully, the pattern can lead to cascading updates, where one change triggers a chain of updates across multiple observers.

- Complexity in Managing Dependencies: As the number of observers increases, managing their dependencies and interactions can become complex.