Observer in Java

Observer is a behavioral design pattern that allows some objects to notify other objects about changes in their state.

The Observer pattern provides a way to subscribe and unsubscribe to and from these events for any object that implements a subscriber interface.

Learn more about Observer →

Navigation

Complexity: ★★☆

Popularity: ★★★

components. It provides a way to react to events happening in other objects without coupling to their classes.

Here are some examples of the pattern in core Java libraries:

- `java.util.Observer` / `java.util.Observable` (rarely used in real world)
- All implementations of `java.util.EventListener` (practically all over Swing components)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

Identification: The pattern can be recognized if you see a subscription method that stores incoming objects in a list. You can confirm the identification, if you see some sort of notification method that iterates over objects in that list and calls their "update" method.

Event subscription

In this example, the Observer pattern establishes indirect collaboration between objects of a text editor. Each time the `Editor` object changes, it notifies its subscribers. `EmailNotificationListener` and `LogOpenListener` react to these notifications by executing their primary behaviors.

Subscriber classes aren't coupled to the editor class and can be reused in other apps if needed. The `Editor` class depends only on the abstract subscriber interface. This allows adding new subscriber types without changing the editor's code.

publisher

publisher/EventManager.java: Basic publisher

```
package refactoring_guru.observer.example.publisher;

import refactoring_guru.observer.example.listeners.EventListener;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
    Map<String, List<EventListener>> listeners = new HashMap<>();

    public EventManager(String... operations) {
        for (String operation : operations) {
            this.listeners.put(operation, new ArrayList<>());
        }
    }

    public void subscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.add(listener);
    }

    public void unsubscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.remove(listener);
    }

    public void notify(String eventType, File file) {
        List<EventListener> users = listeners.get(eventType);
        for (EventListener listener : users) {
            listener.update(eventType, file);
        }
    }
}
```

editor

editor/Editor.java: Concrete publisher, tracked by other objects

```
package refactoring_guru.observer.example.editor;

import refactoring_guru.observer.example.publisher.EventManager;

import java.io.File;

public class Editor {
    public EventManager events;
    private File file;

    public Editor() {
        this.events = new EventManager("open", "save");
    }

    public void openFile(String filePath) {
        this.file = new File(filePath);
```

```
    public void saveFile() throws Exception {
        if (this.file != null) {
            events.notify("save", file);
        } else {
            throw new Exception("Please open a file first.");
        }
    }
}
```

listeners

listeners/EventListener.java: Common observer interface

```
package refactoring_guru.observer.example.listeners;

import java.io.File;

public interface EventListener {
    void update(String eventType, File file);
}
```

listeners/EmailNotificationListener.java: Sends emails upon receiving notification

```
package refactoring_guru.observer.example.listeners;

import java.io.File;

public class EmailNotificationListener implements EventListener {
    private String email;

    public EmailNotificationListener(String email) {
        this.email = email;
    }

    @Override
    public void update(String eventType, File file) {
        System.out.println("Email to " + email + ": Someone has performed " + eventType +
    }
}
```

```
package refactoring_guru.observer.example.listeners;

import java.io.File;

public class LogOpenListener implements EventListener {
    private File log;

    public LogOpenListener(String fileName) {
        this.log = new File(fileName);
    }

    @Override
    public void update(String eventType, File file) {
        System.out.println("Save to log " + log + ": Someone has performed " + eventType
    }
}
```

Demo.java: Initialization code

```
package refactoring_guru.observer.example;

import refactoring_guru.observer.example.editor.Editor;
import refactoring_guru.observer.example.listeners.EmailNotificationListener;
import refactoring_guru.observer.example.listeners.LogOpenListener;

public class Demo {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.events.subscribe("open", new LogOpenListener("/path/to/log/file.txt"));
        editor.events.subscribe("save", new EmailNotificationListener("admin@example.com"

        try {
            editor.openFile("test.txt");
            editor.saveFile();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

OutputDemo.txt: Execution result