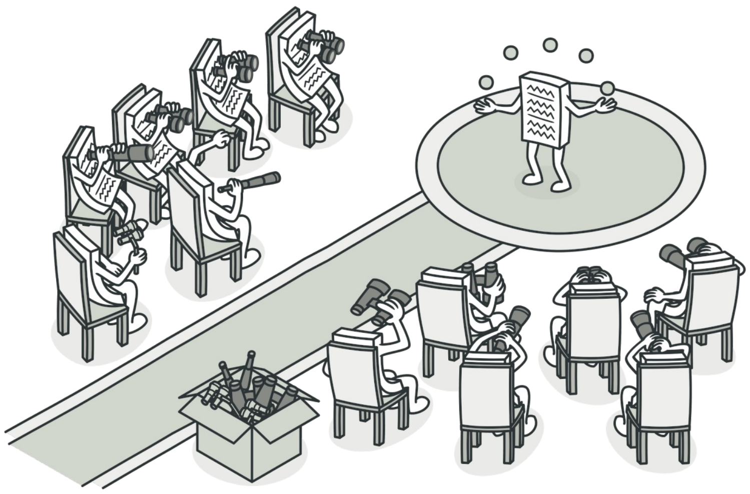
Advanced Software Engineering Part 08 – Observer Pattern

Dr. Amjad AbuHassan

Intent

- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Problem

- Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it’s a new model of the iPhone) which should become available in the store very soon.
- The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

Problem cont.

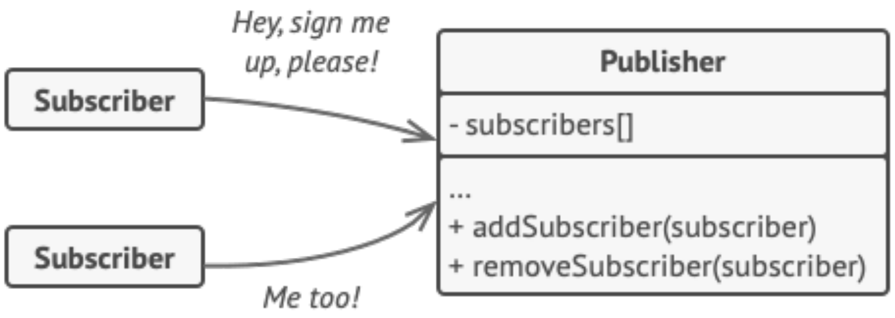
- On the other hand, the store could send tons of emails to all customers each time a new product becomes available.
  - This would save some customers from endless trips to the store.
  - At the same time, it’d upset other customers who aren’t interested in new products.
- It looks like we’ve got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

Solution

- The object that has some interesting state is often called subject, but since it’s also going to notify other objects about the changes to its state, we’ll call it publisher.
- All other objects that want to track changes to the publisher’s state are called subscribers.

Solution cont.

- The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

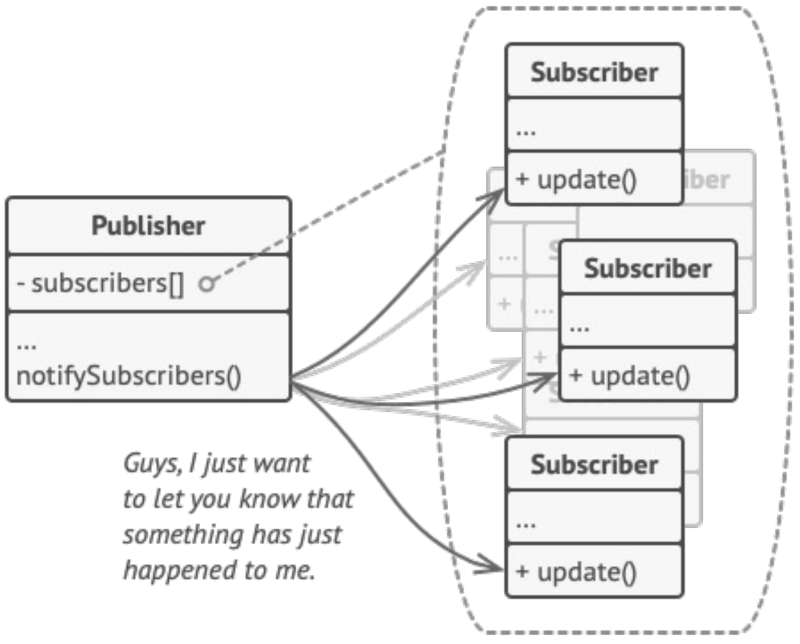


Solution cont.

- Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class.
    - You wouldn't want to couple the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.
- So it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface.
    - This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.

Solution cont.

Publisher notifies subscribers by calling the specific notification method on their objects.

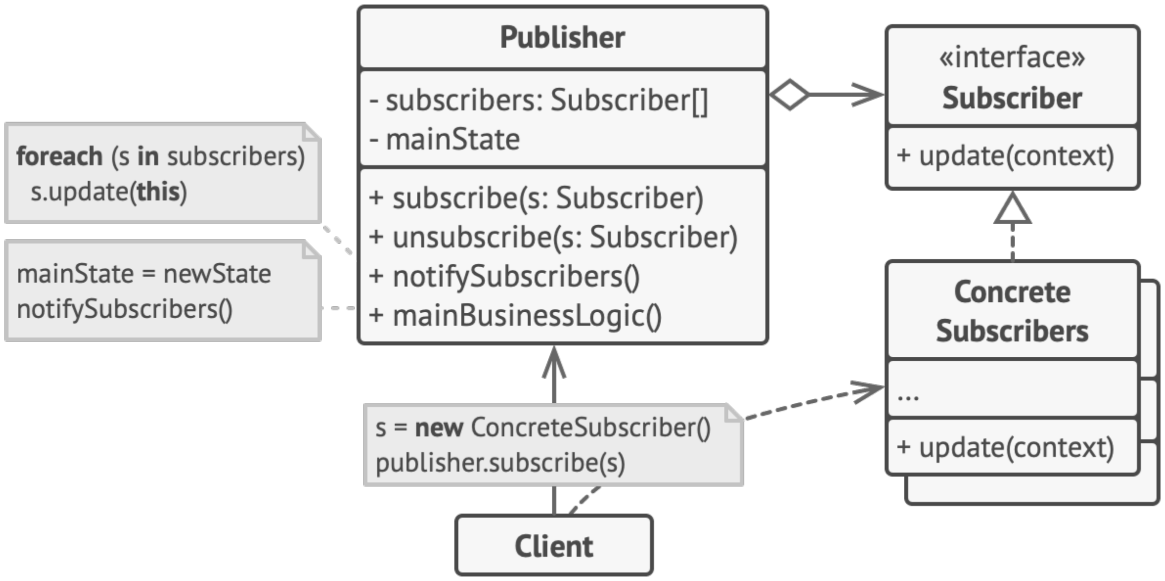


Real-World Analogy

If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

Structure



Pseudocode

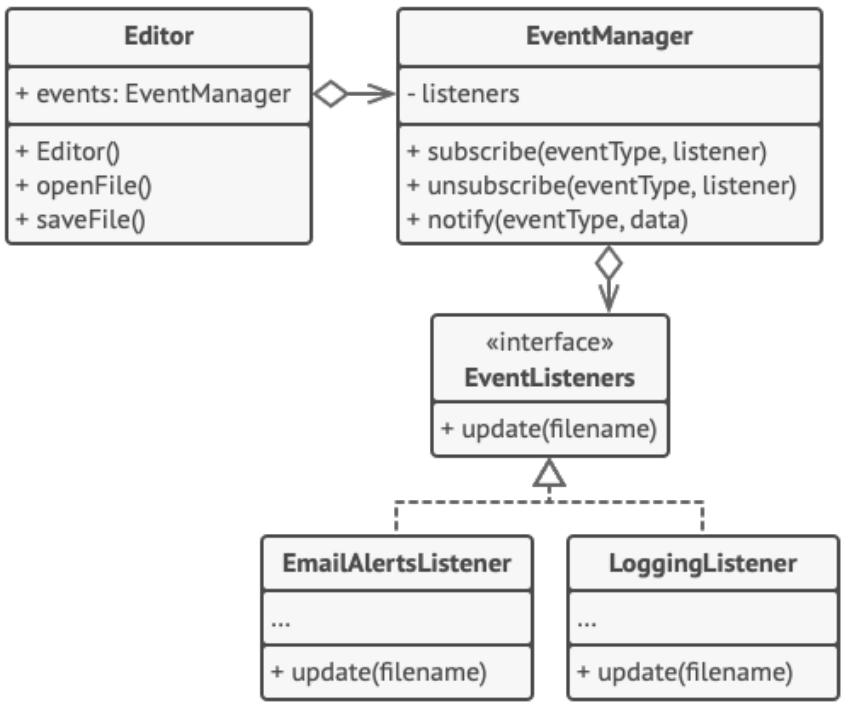
In this example,

the **Observer** pattern lets

the text editor object notify

other service objects about

changes in its state.



Pseudocode

```
public class EventManager {
  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);
    }
  }
}
```

Pseudocode cont.

```
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);
    events.notify("open", file);
  }
```

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

Pseudocode cont.

```
public interface EventListener {
  void update(String eventType, File 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 + " operation with the following file: " + file.getName());
  }
}
```

Pseudocode cont.

```
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 + " operation with the following file: " + file.getName());
  }
}
```

Pseudocode cont.

```
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();
    }
  }
}
```

```
Save to log \path\to\log\file.txt: Someone has performed open operation with the following file: test.txt
Email to admin@example.com: Someone has performed save operation with the following file: test.txt
```

Applicability

Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

- You can often experience this problem when working with classes of the graphical user interface. For example, you created custom button classes, and you want to let the clients hook some custom code to your buttons so that it fires whenever a user presses a button.
- The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects. You can add the subscription mechanism to your buttons, letting the clients hook up their custom code via custom subscriber classes.

Applicability cont.

Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

- The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.