

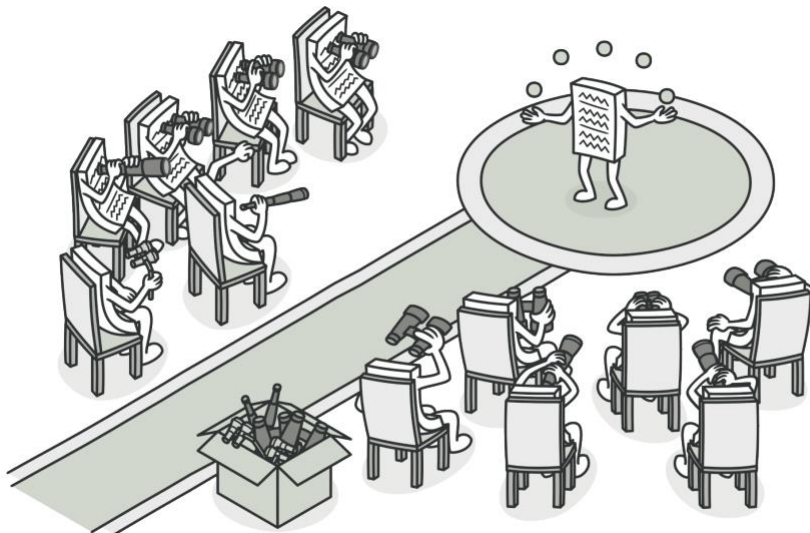
Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif

1



OBSERVER

2



Observer

- *Also known as: Event-Subscriber, Listener*
- **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.

3



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.

On the other hand, the store could send tons of emails (which might be considered spam) 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.

4

Problem

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.

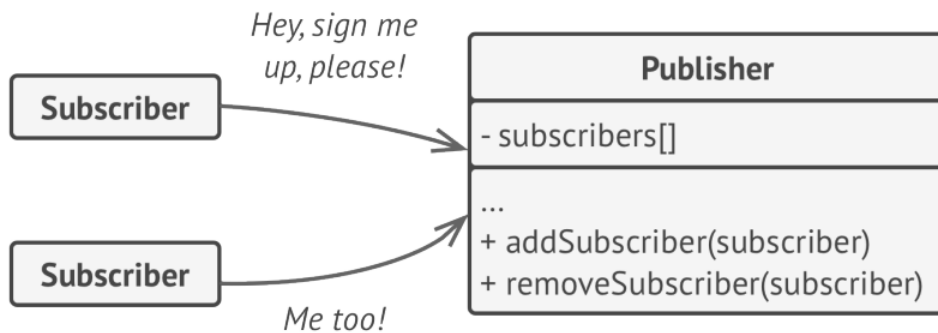
5

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*.

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. Fear not! Everything isn't as complicated as it sounds. In reality, this mechanism consists of 1) an array field for storing a list of references to subscriber objects and 2) several public methods which allow adding subscribers to and removing them from that list.

6

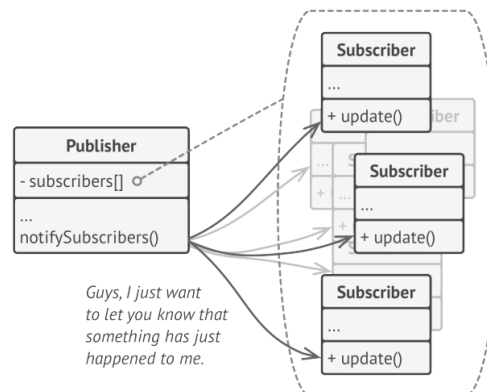


A subscription mechanism lets individual objects subscribe to event notifications.

7

Solution

Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.



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

8

Solution

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. That's why it's crucial that all subscribers implement the same interface and that the publisher communicate 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.

9

Solution

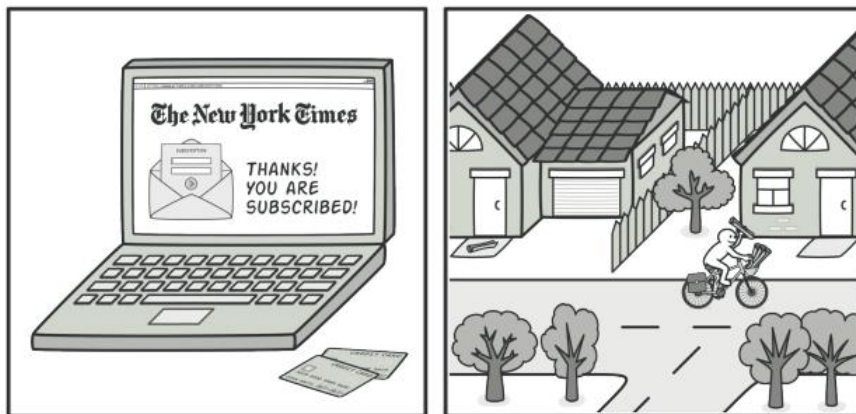
If your app has several different types of publishers and you want to make your subscribers compatible with all of them, you can go even further and make all publishers follow the same interface. This interface would only need to describe a couple of the subscription methods. The interface would allow subscribers to observe publishers' states without **coupling** to their concrete classes.

10

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.

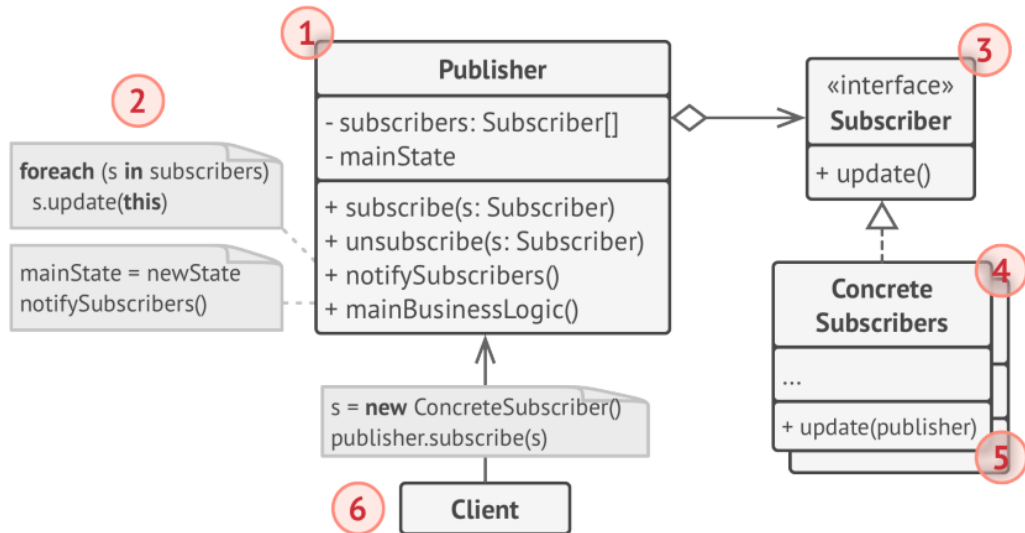
11



Magazine and newspaper subscriptions.

12

Structure



13

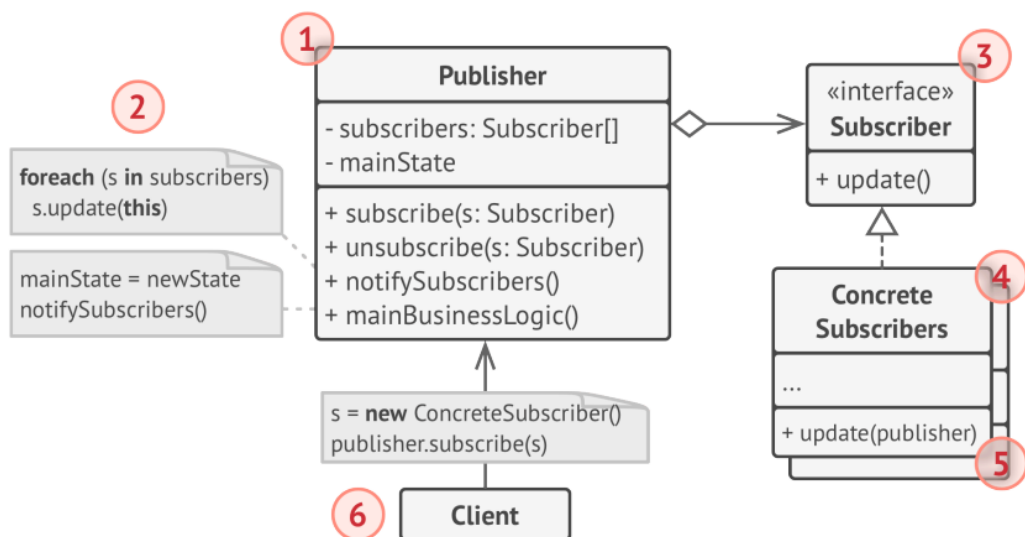
- The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.
- When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.
- The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.

14

- **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.
- Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.
- The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

15

Structure

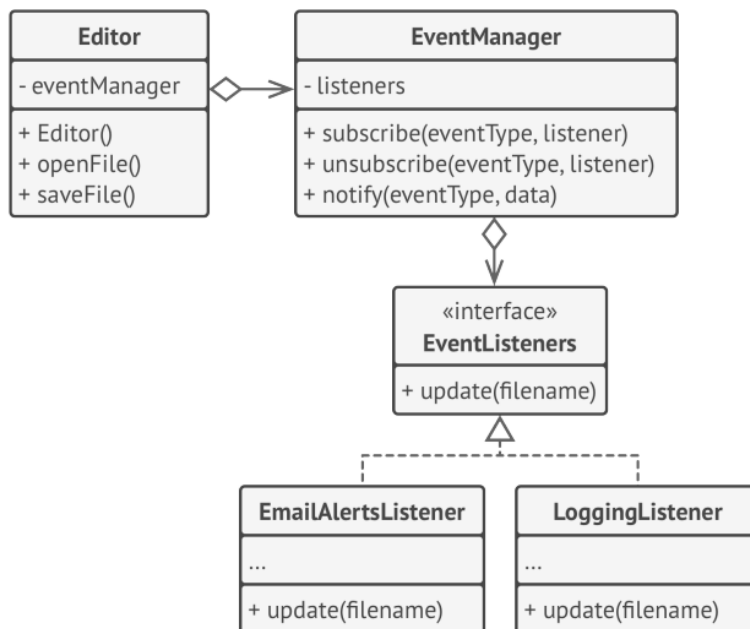


16

Pseudocode


- In this example, the **Observer** pattern lets the text editor object notify other service objects about changes in its state.
- The list of subscribers is compiled dynamically: objects can start or stop listening to notifications at runtime, depending on the desired behavior of your app.
- In this implementation, the editor class doesn't maintain the subscription list by itself. It delegates this job to the special helper object devoted to just that.
- Adding new subscribers to the program doesn't require changes to existing publisher classes, as long as they work with all subscribers through the same interface.

17




Notifying objects about events that happen to other objects.

18



```
1 // The base publisher class includes subscription management
2 // code and notification methods.
3 class EventManager is
4 private field listeners: hash map of event types and listeners
5
6 method subscribe(eventType, listener) is
7 listeners.add(eventType, listener)
8
9 method unsubscribe(eventType, listener) is
10 listeners.remove(eventType, listener)
11
12 method notify(eventType, data) is
13 foreach (listener in listeners.of(eventType)) do
14 listener.update(data)
```

19



```
22 class Editor is
23 private field events: EventManager
24 private field file: File
25
26 constructor Editor() is
27 events = new EventManager()
28 // Methods of business logic can notify subscribers about
29 // changes.
30 method openFile(path) is
31 this.file = new File(path)
32 events.notify("open", file.name)
33
34 method saveFile() is
35 file.write()
36 events.notify("save", file.name)
37
38 // ...
39
```

20

```

40
41 // Here's the subscriber interface. If your programming language
42 // supports functional types, you can replace the whole
43 // subscriber hierarchy with a set of functions.
44 interface EventListener is
45 method update(filename)
46
47 // Concrete subscribers react to updates issued by the publisher
48 // they are attached to.
49 class LoggingListener is
50 private field log: File
51 private field message
52
53 constructor LoggingListener(log_filename, message) is
54 this.log = new File(log_filename)
55 this.message = message
56
57 method update(filename) is
58 log.write(replace('%s',filename,message))
59

```


21

```

60 class EmailAlertsListener is
61 private field email: string
62
63 constructor EmailAlertsListener(email, message) is
64 this.email = email
65 this.message = message
66
67 method update(filename) is
68 system.email(email, replace('%s',filename,message))
69
70

```

22



```
71 // An application can configure publishers and subscribers at
72 // runtime.
73 class Application is
74 method config() is
75 editor = new TextEditor()
76
77 logger = new LoggingListener(
78 "/path/to/log.txt",
79 "Someone has opened the file: %s");
80 editor.events.subscribe("open", logger)
81
82 emailAlers = new EmailAlertsListener(
83 "admin@example.com",
84 "Someone has changed the file: %s")
85 editor.events.subscribe("save", emailAlers)
```

23



Java Code Example

24

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

How to Implement

1. Look over your business logic and try to break it down into two parts: the core functionality, independent from other code, will act as the publisher; the rest will turn into a set of subscriber classes.
2. Declare the subscriber interface. At a bare minimum, it should declare a single update method.
3. Declare the publisher interface and describe a pair of methods for adding a subscriber object to and removing it from the list. Remember that publishers must work with subscribers only via the subscriber interface.

27

How to Implement

4. Decide where to put the actual subscription list and the implementation of subscription methods. Usually, this code looks the same for all types of publishers, so the obvious place to put it is in an abstract class derived directly from the publisher interface. Concrete publishers extend that class, inheriting the subscription behavior.
5. Create concrete publisher classes. Each time something important happens inside a publisher, it must notify all its subscribers.

28

How to Implement

6. Implement the update notification methods in concrete subscriber classes. Most subscribers would need some context data about the event. It can be passed as an argument of the notification method. But there's another option. Upon receiving a notification, the subscriber can fetch any data directly from the notification. In this case, the publisher must pass itself via the update method. The less flexible option is to link a publisher to the subscriber permanently via the constructor.
7. The client must create all necessary subscribers and register them with proper publishers.

29

Pros and Cons

- *Open/Closed Principle*. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- You can establish relations between objects at runtime.
- Subscribers are notified in random order.

30



Review