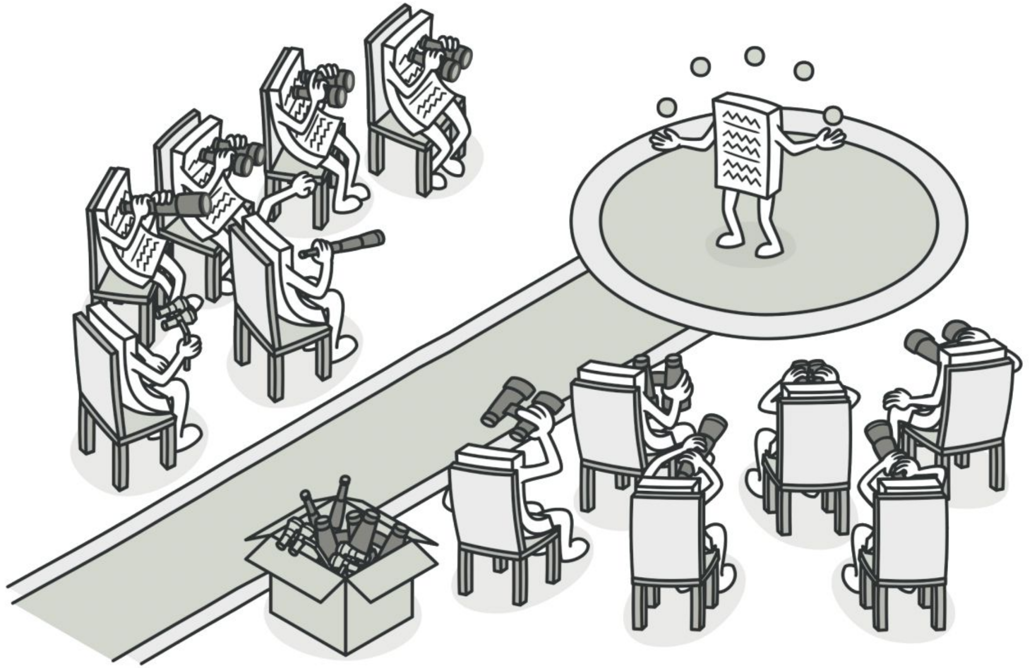
Lecture 7: Design Patterns

Observer

Observer AKA Event-Subscriber AKA listener



Observer AKA Event-Subscriber AKA listener

Observer: 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

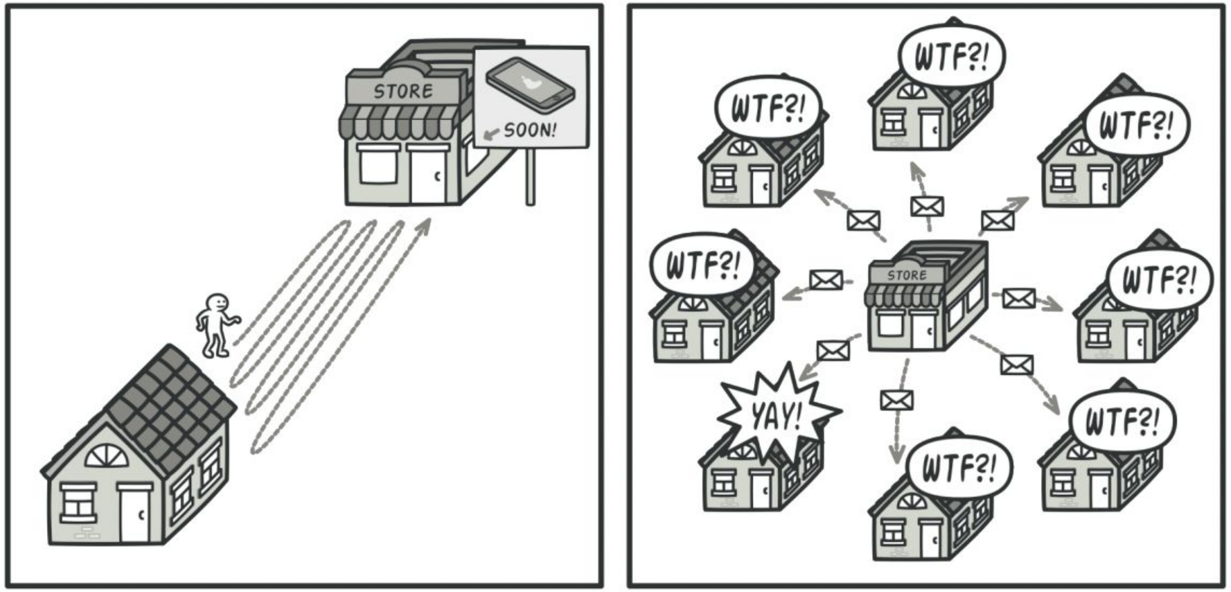
these trips would be pointless.

Observer: Problem

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.

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.

Observer: Problem



Visiting the store vs. sending spam

Observer: 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

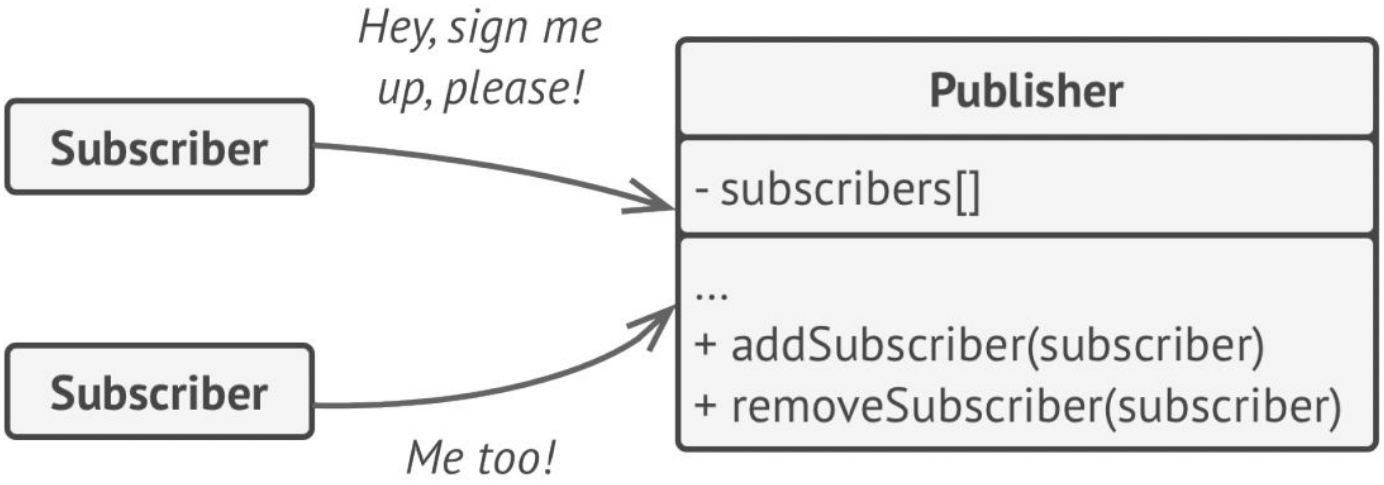
subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn't as complicated

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.

Observer: Solution



A subscription mechanism lets individual objects subscribe to event notifications.

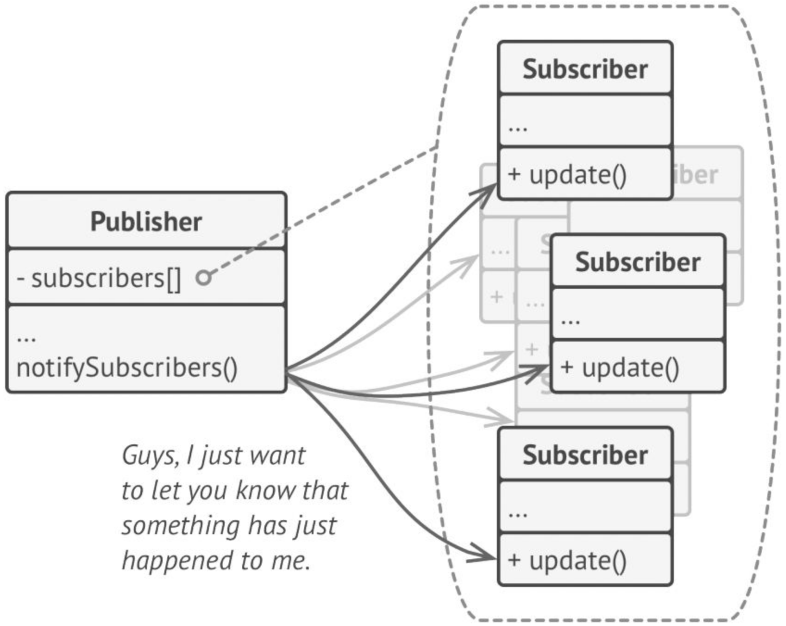
Observer: Solution

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

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.

That’s why it’s crucial that all subscribers implement the same interface and that the publisher communicates with them only

Observer: Solution



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

Observer: 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.

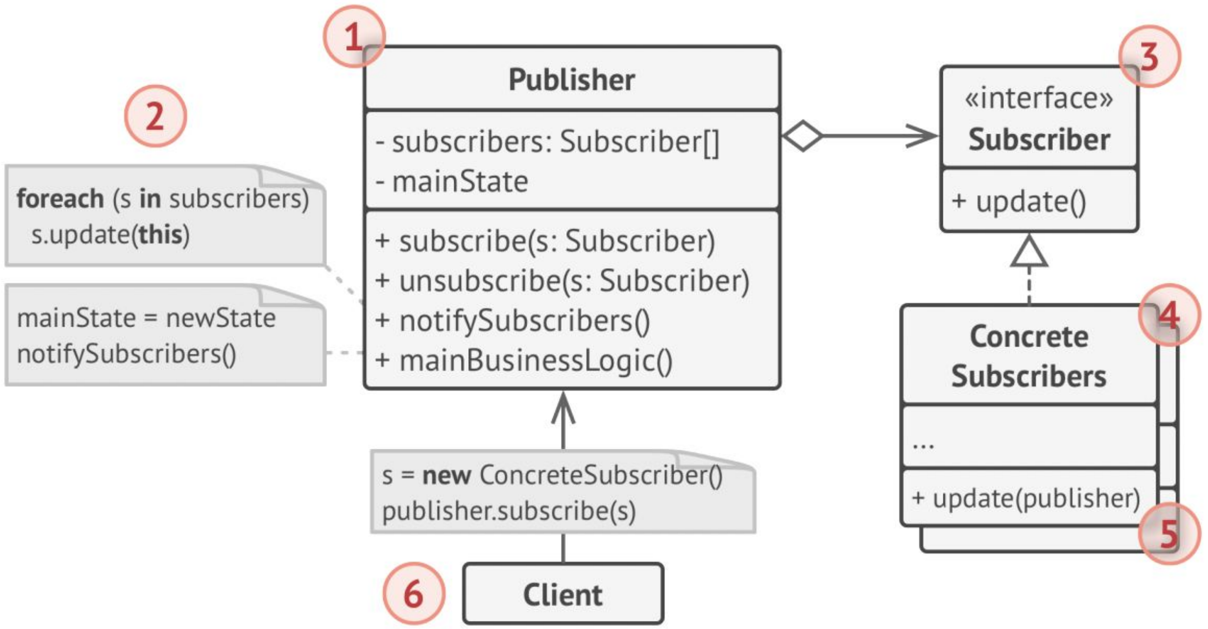
The publisher maintains a list of subscribers and knows which magazines they’re interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

Observer: Real World Analogy

HTTPS://WWW.NYTIMES.COM/SUBSCRIPTION/
The New York Times
SUBSCRIPTION
THANKS! YOU ARE SUBSCRIBED!
CREDIT CARD
CREDIT CARD

Magazine and newspaper subscriptions.

Observer: Structure



Observer: Structure

1. 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 infrastruc-
ture that lets new subscribers join and current subscribers

leave the list.

2. When a new event happens, the publisher goes over the sub-
scription list and calls the notification method declared in the

Observer: How to implement

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.

Observer: 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.