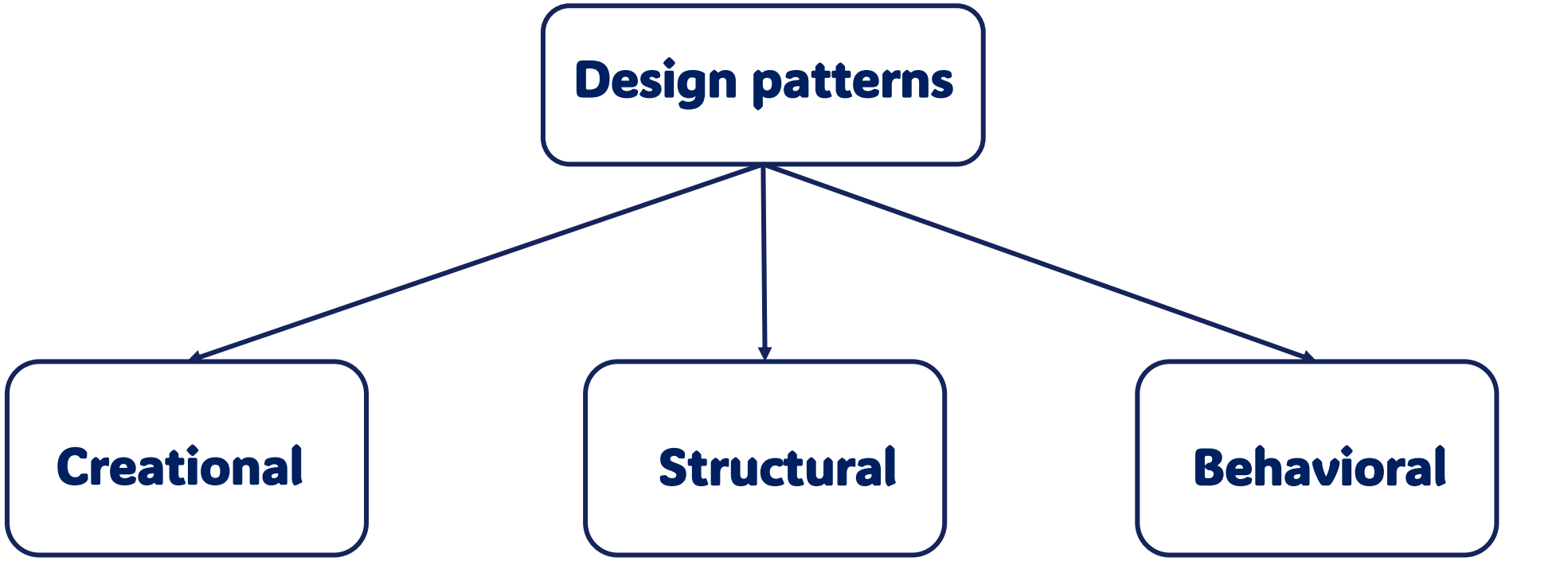
الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Software system design – practical

Lecture 07 – observer design pattern

Eng. Raghad al-hossny

Design patterns:



- ***Factory method***
- ***Singelton***

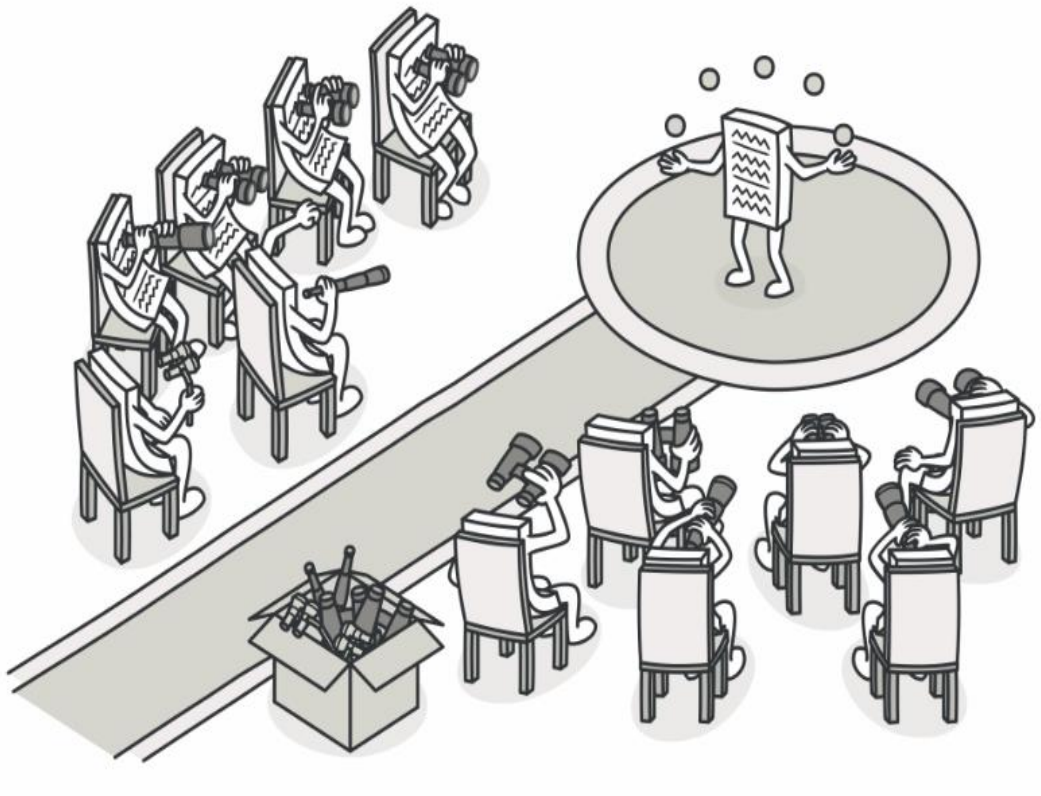
- ***Decorator***

- ***Strategy.***
- ***State.***
- ***Template method.***
- ***observer***

Observer design pattern:

Also known as event-subscriber and listener

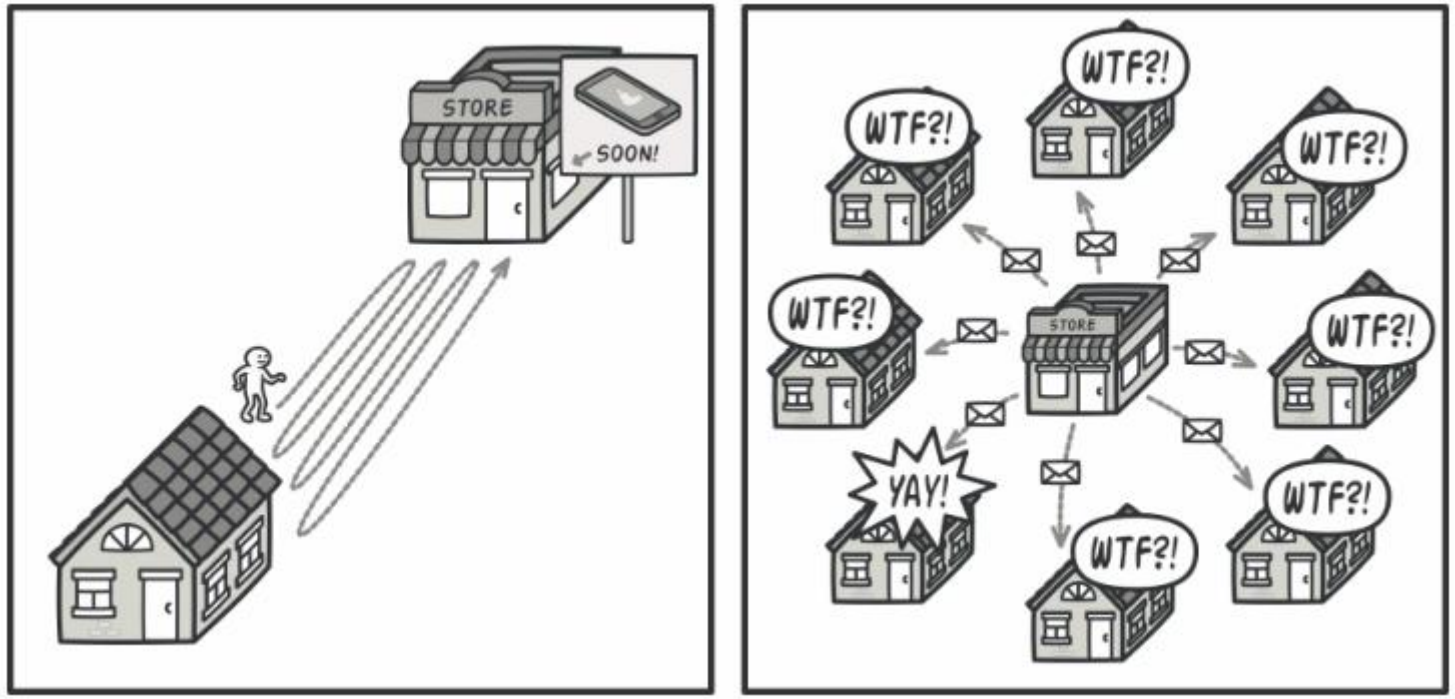
is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Observer design pattern - example:

Imagine that you have two types of objects: a **Customer** and a **Store**.

The customer is very interested in a particular brand of product, which should become available in the store very soon.

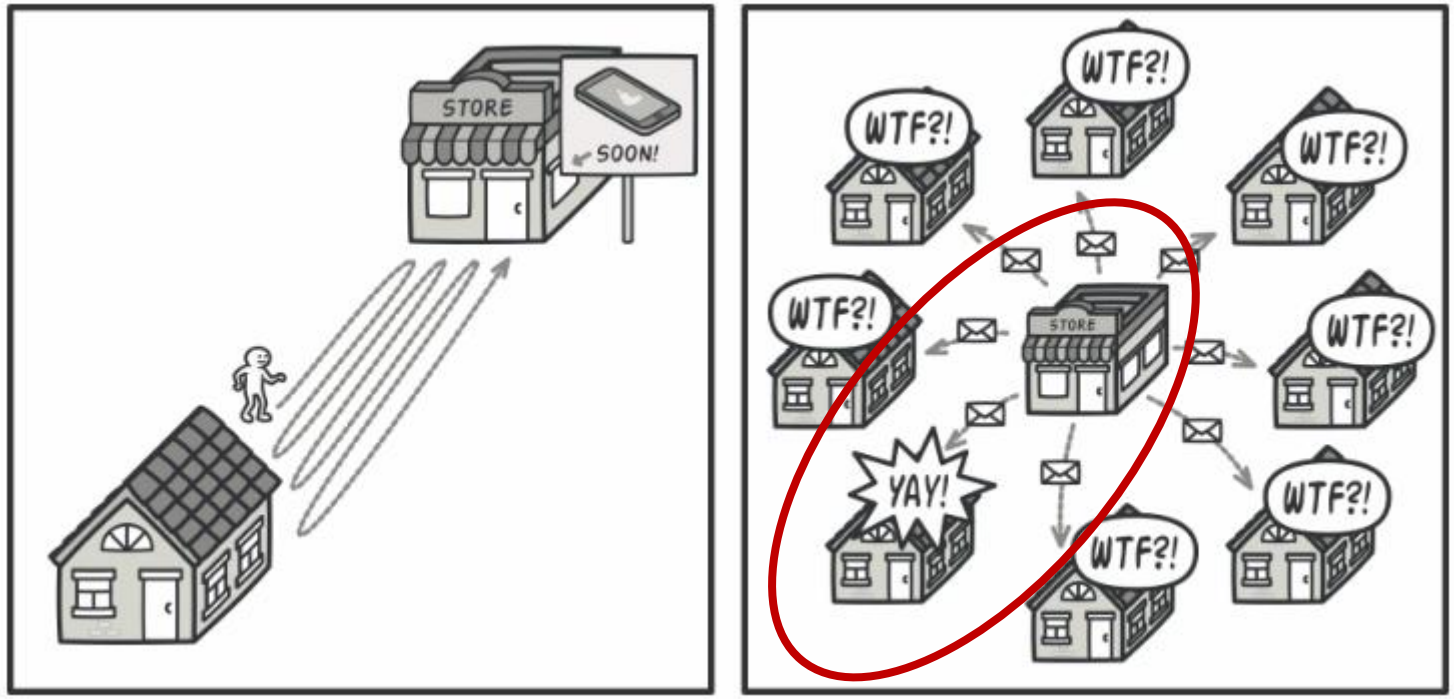


It looks like we've got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

Observer design pattern - example:

The solution:
Notify only the interested clients about the new product arriving.

Same idea as the observer design pattern.

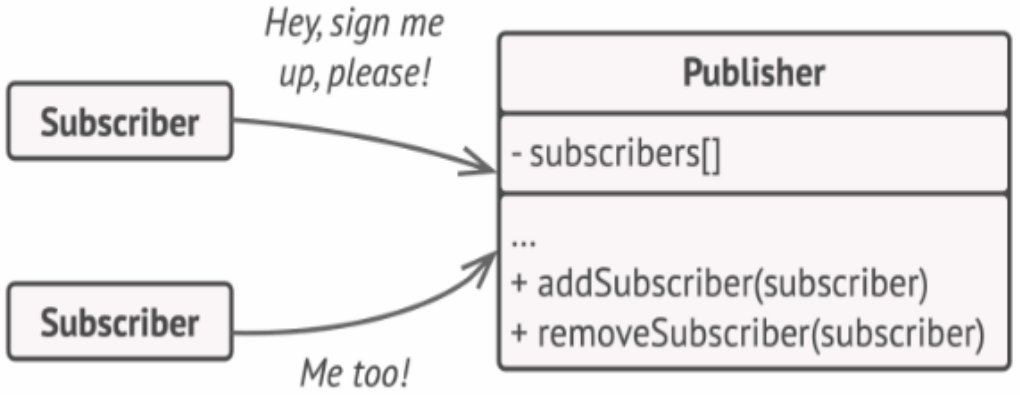


Observer design pattern:

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.

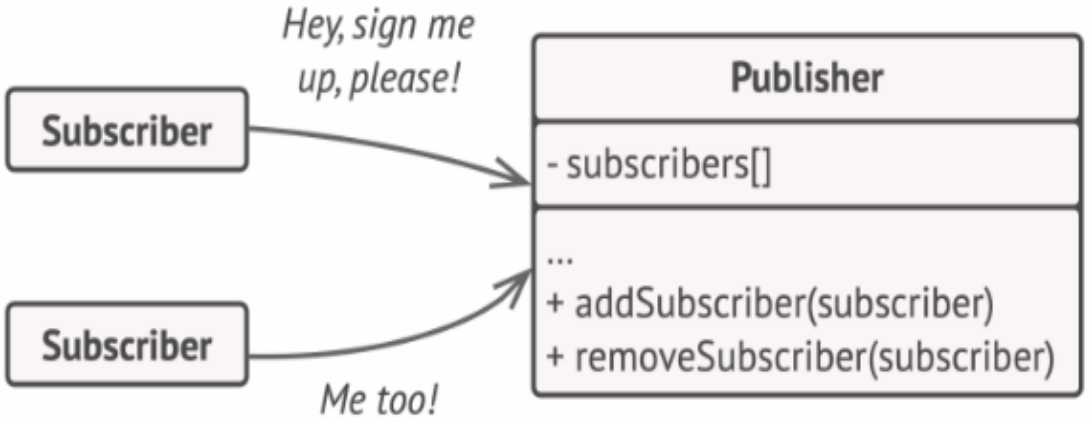


Observer design pattern:

How?

1) an array field for storing a list of references to subscriber objects.

2) several public methods which allow adding subscribers to and removing them from that list.

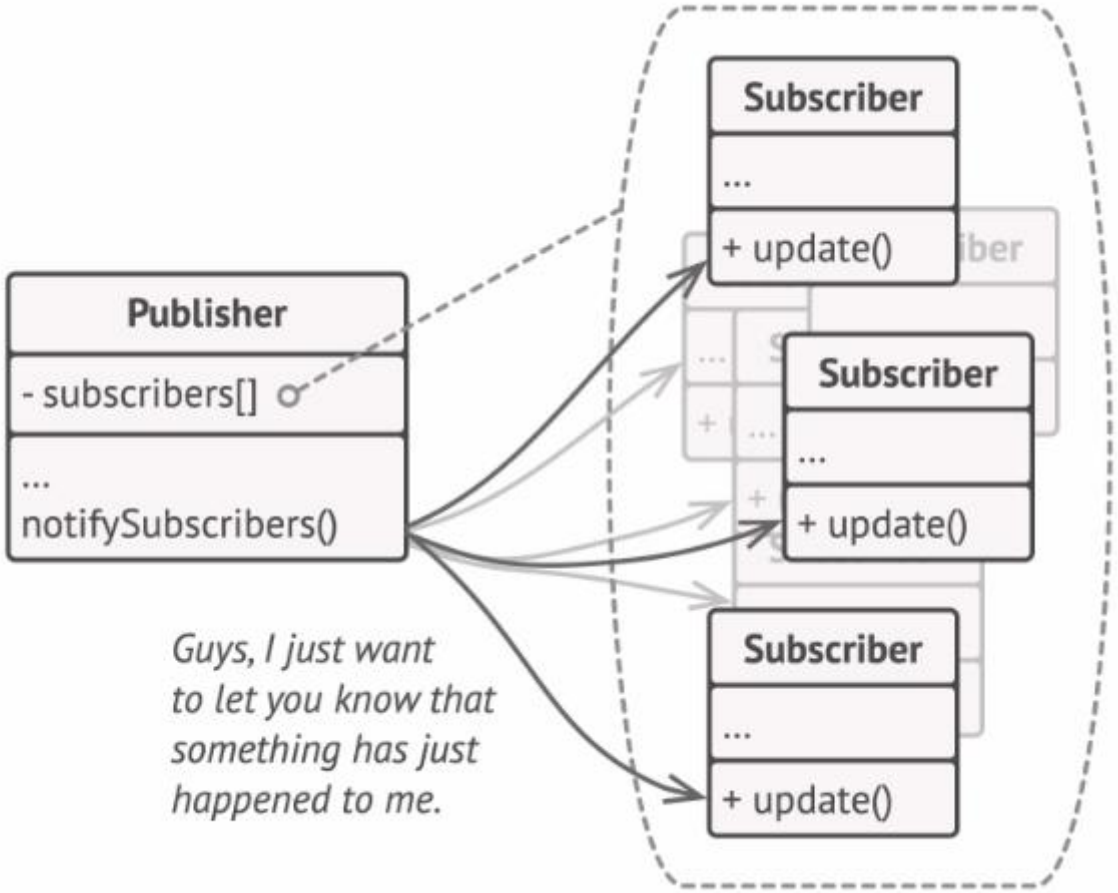


Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

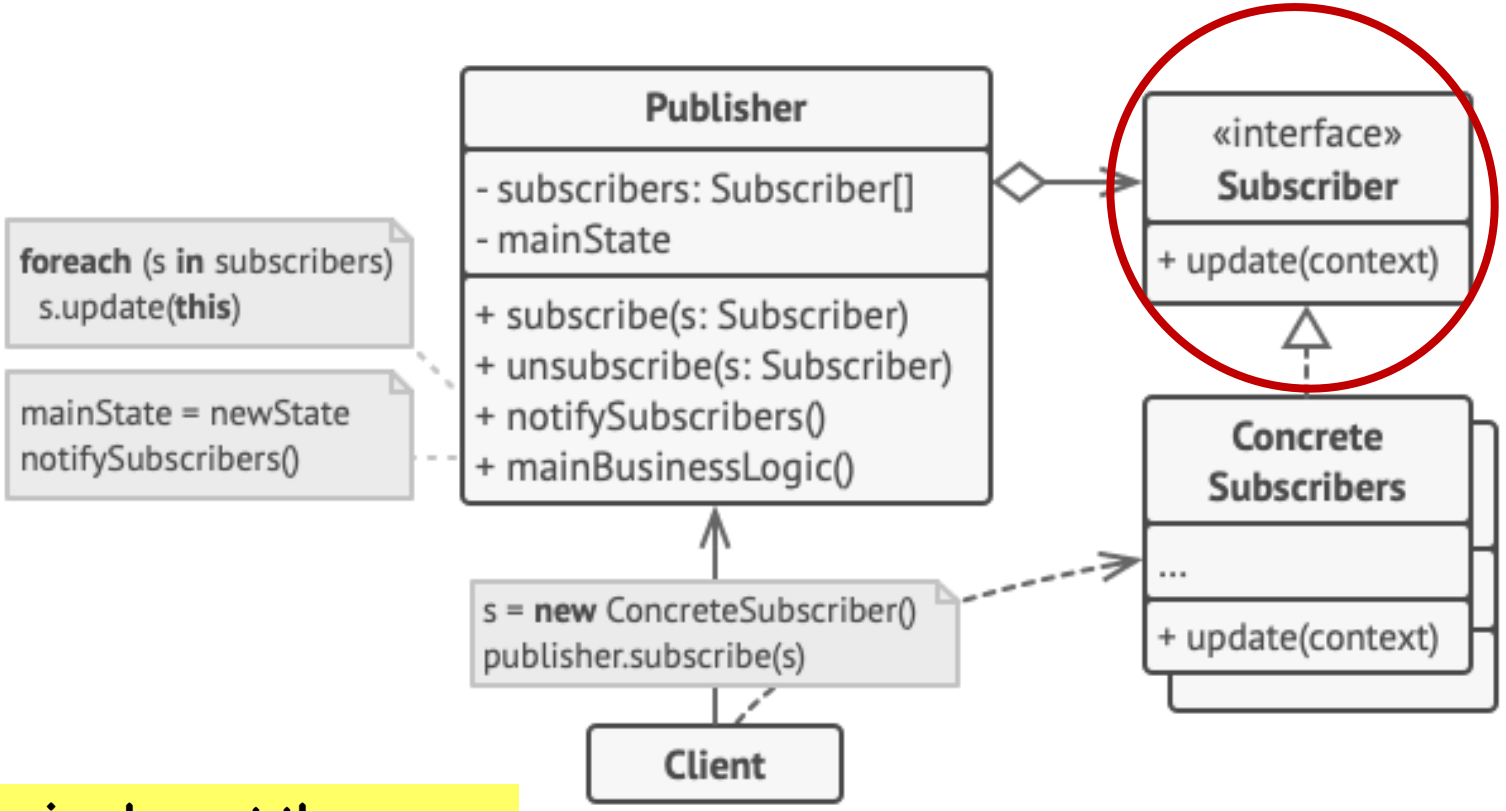
Observer design pattern:

Problem:

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 !!



Observer design pattern:

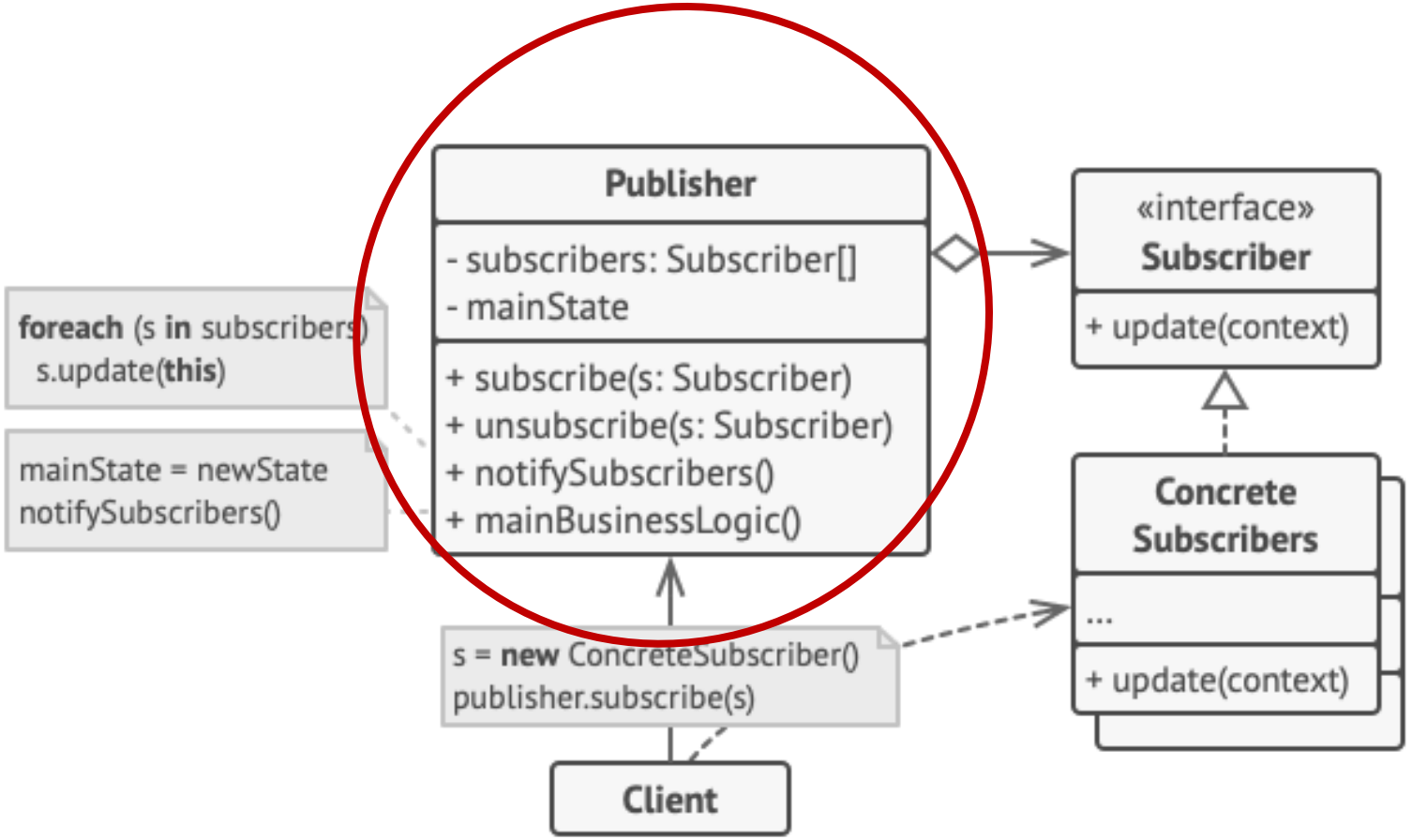


That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface.

Observer design pattern:

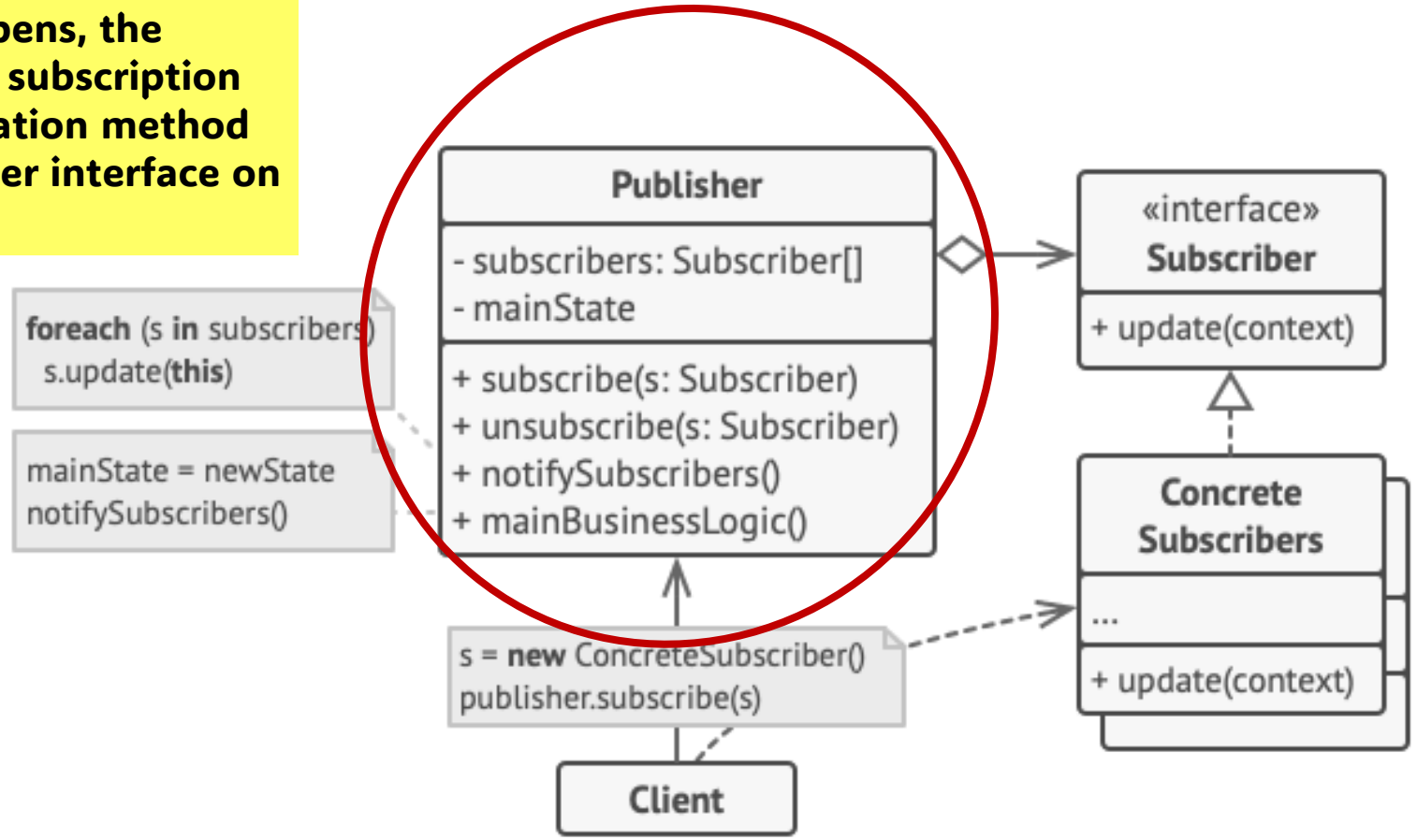
The Publisher issues events of interest to other objects.

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.

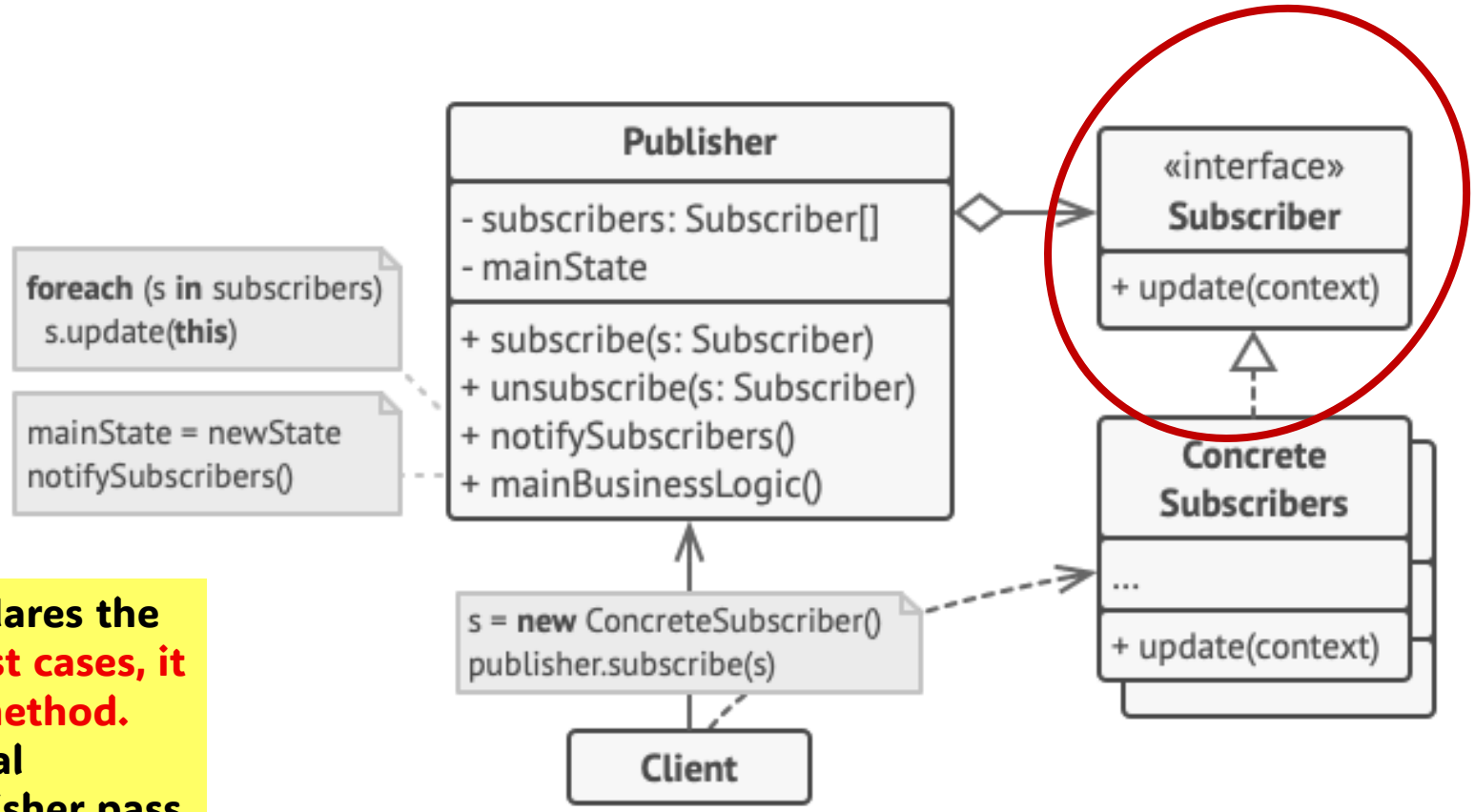


Observer design pattern:

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.

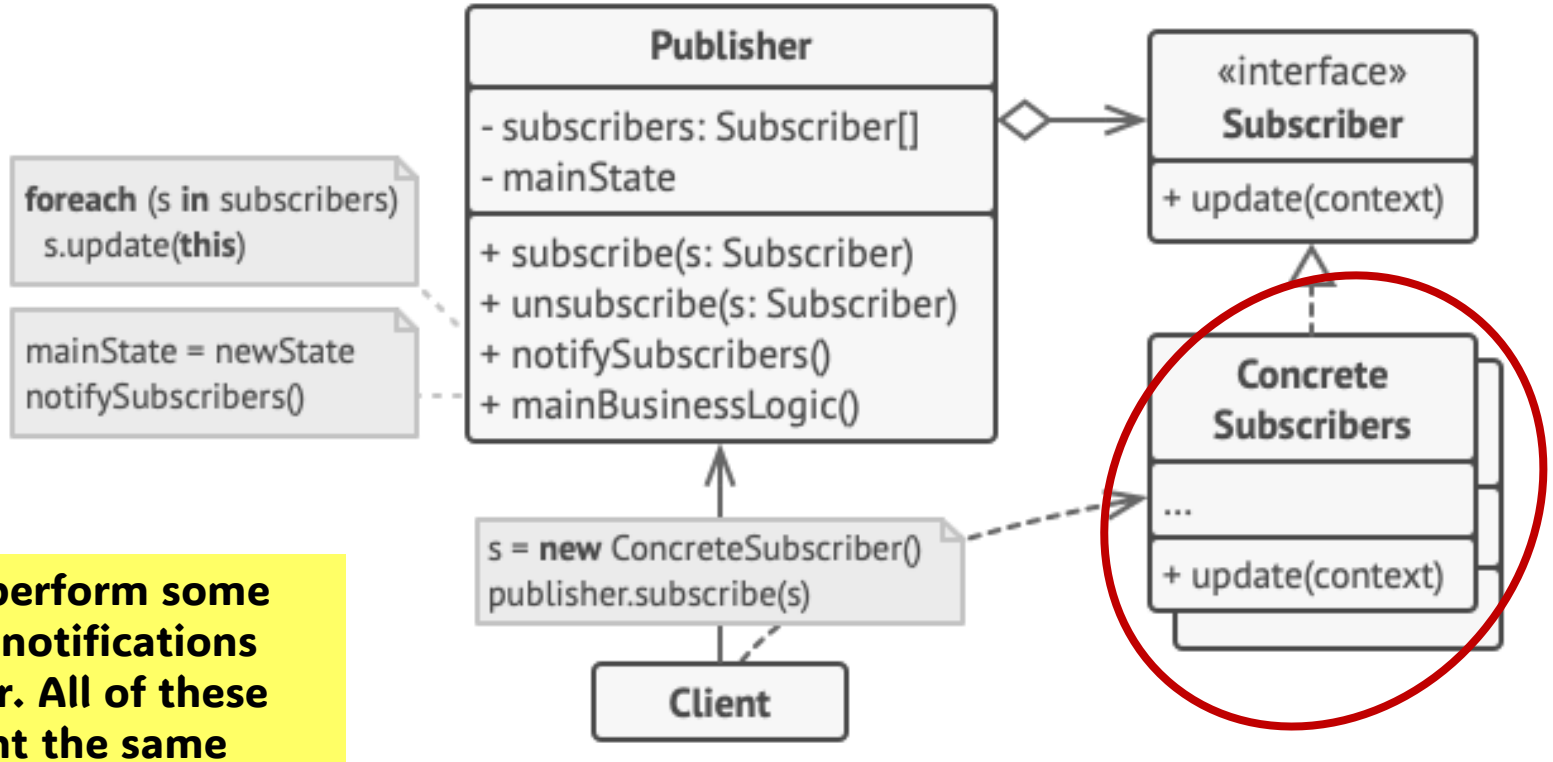


Observer design pattern:



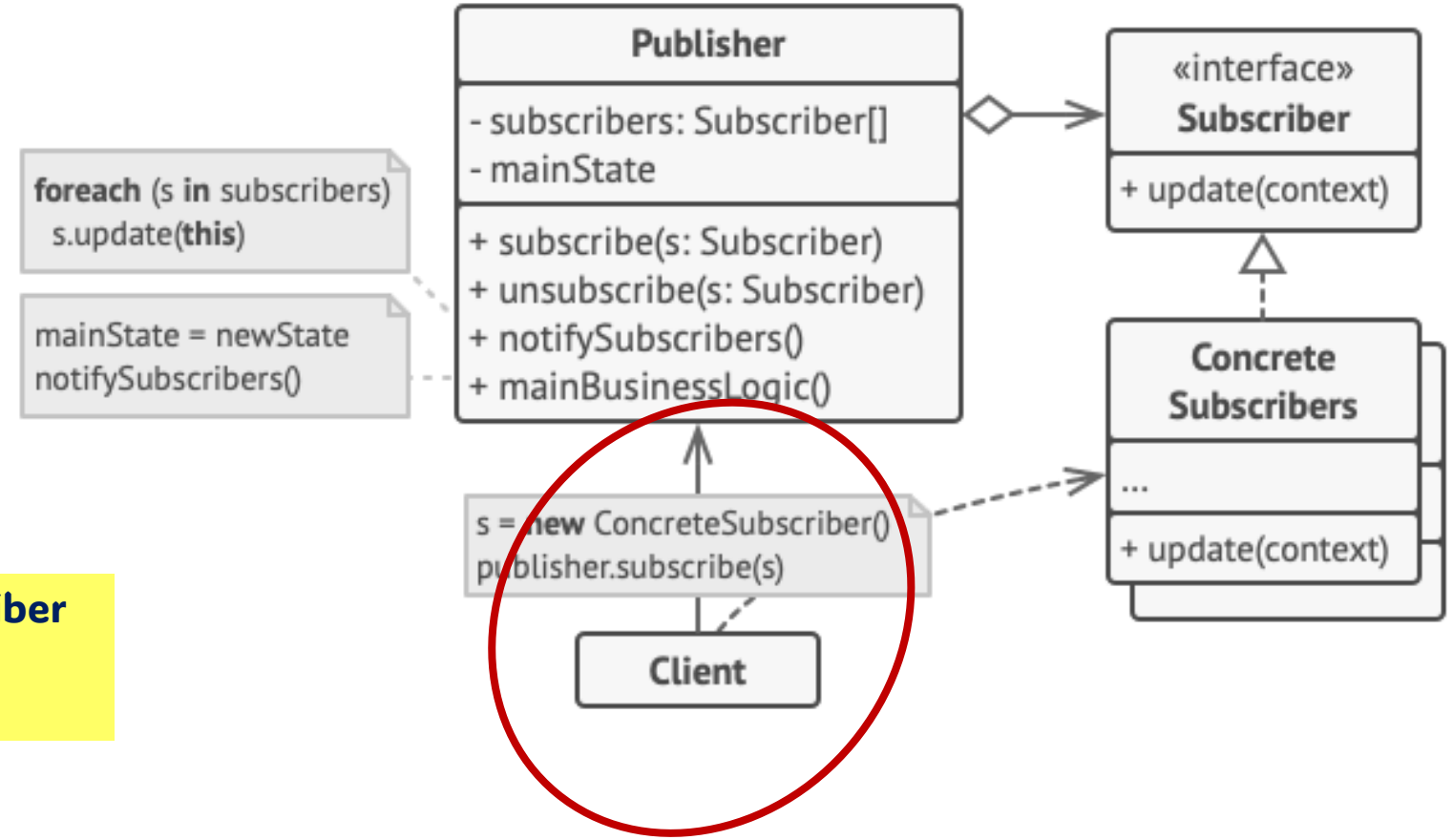
The Subscriber interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.

Observer design pattern:



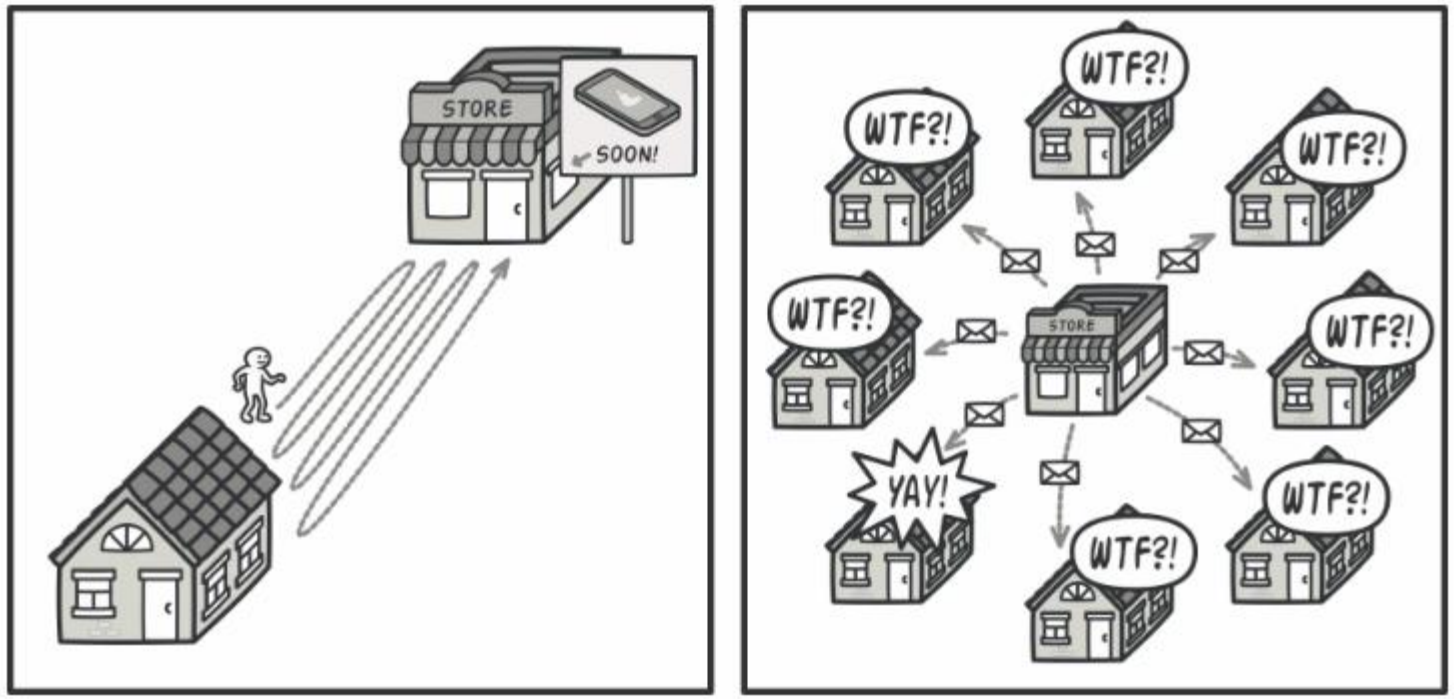
Concrete Subscribers perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

Observer design pattern:



The Client creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

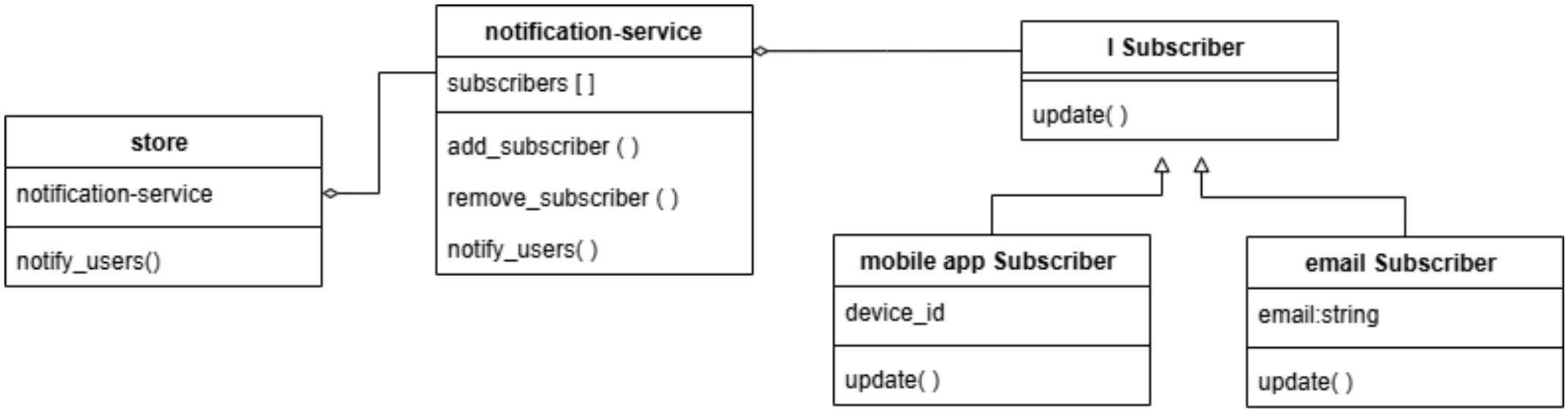
Store example:



Design and implement the store solution using the observer design pattern.

And what if we want a mobile app notification?

Store example - design:



Store example - implementation:

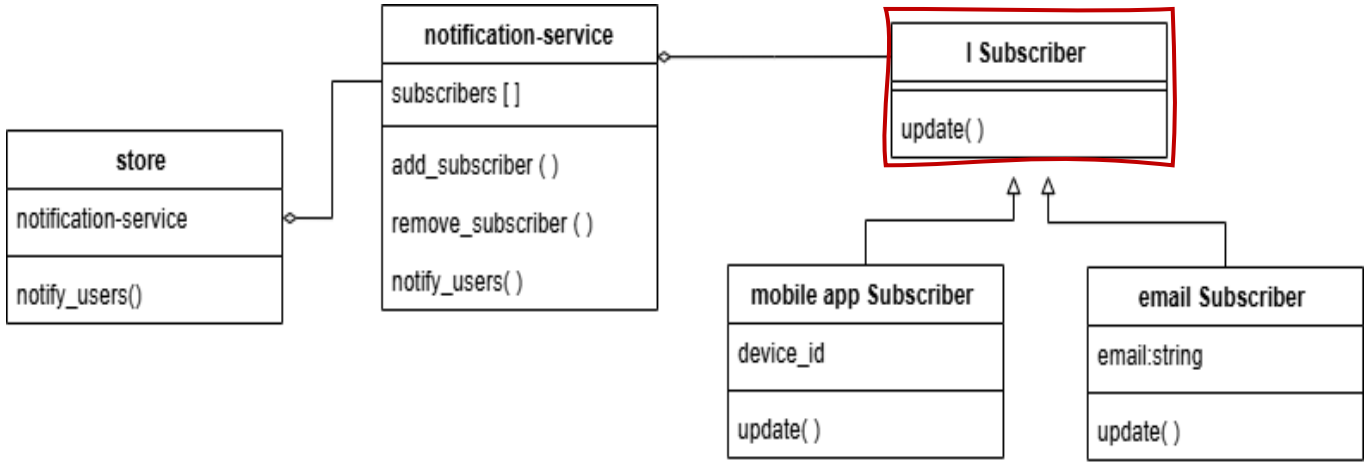
```
package observer;

public interface EventListener {

        void update(String message);


}
```

The main interface:
the observer interface.



Store example - implementation:

```java
public class MobileAppListener implements EventListener {

    private final String username;

    public MobileAppListener(String username) {
        this.username = username;
    }

    @Override
    public void update(String message) {
        System.out.println("Push notification sent to " + username + ": " + message);
    }
}
```

The concrete observers.

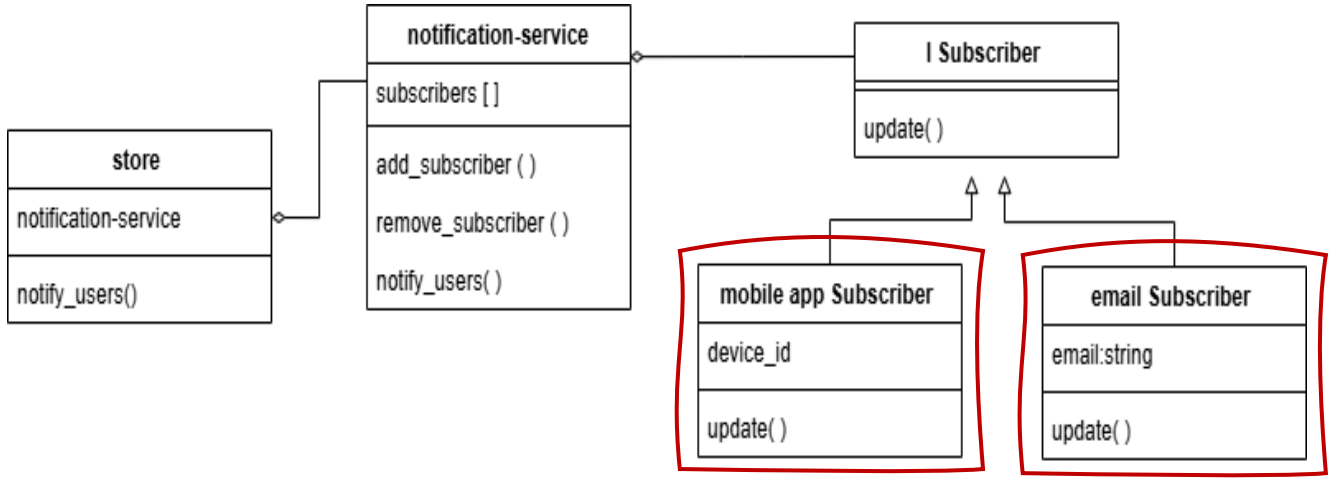
```java
public class EmailMsgListener implements EventListener {

    private final String email;

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

    @Override
    public void update(String message) {
        System.out.println("Email sent to " + email + ": " + message);
    }

}
```



Store example - implementation:

```
public class NotificationService {

    private final List<EventListener> listeners = new ArrayList<>();

    public void subscribe(EventListener listener) {
        listeners.add(listener);
    }

    public void unsubscribe(EventListener listener) {
        listeners.remove(listener);
    }

    public void notifyAllUsers(String message) {
        for (EventListener listener : listeners) {
            listener.update(message);
        }
    }
}
```

The publisher class:

defines the subscribers' list and notifies them.