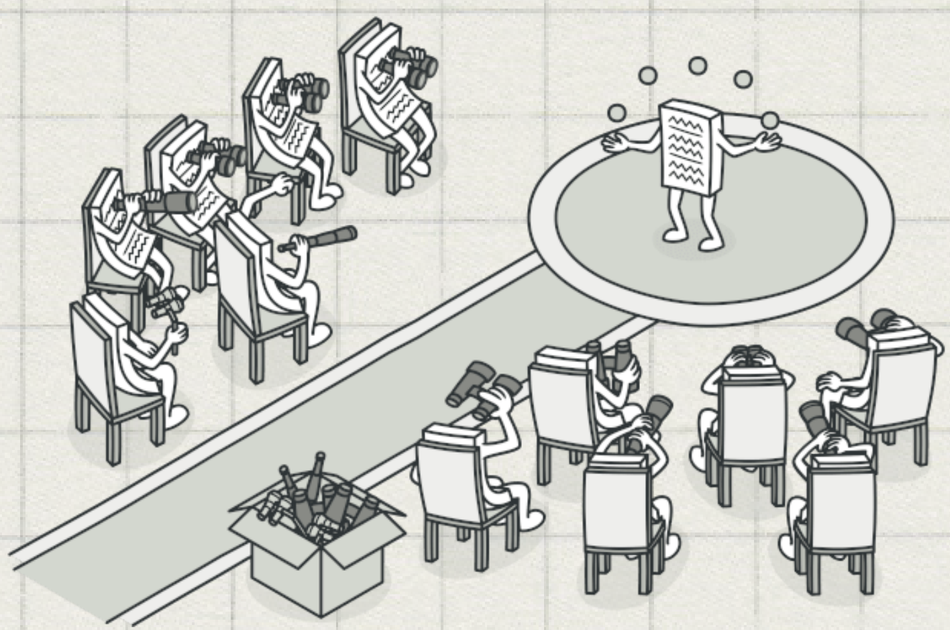
THE OBSERVER DESIGN PATTERN

Apavaloaei Alexandru-Teodor,
Boda Viktoria, Brinza Alina-Elena

OBSERVER (EVENT-SUBSCRIBER, LISTENER)

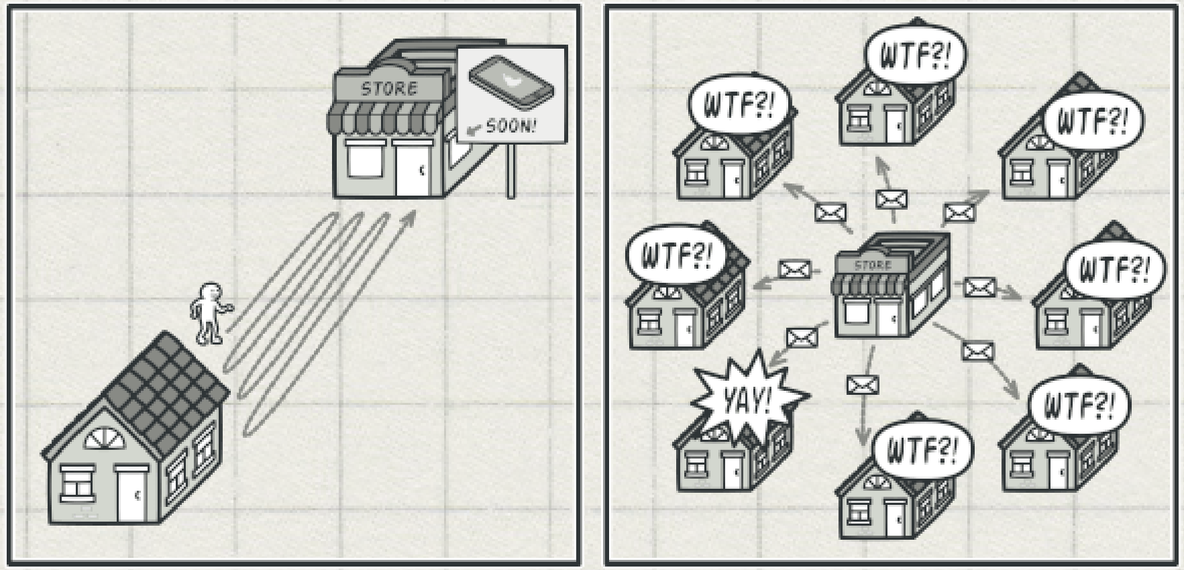
- behavioral design pattern
- allows defining a subscription mechanism to notify multiple objects about any events that happen to the object they're observing



Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

PROBLEM

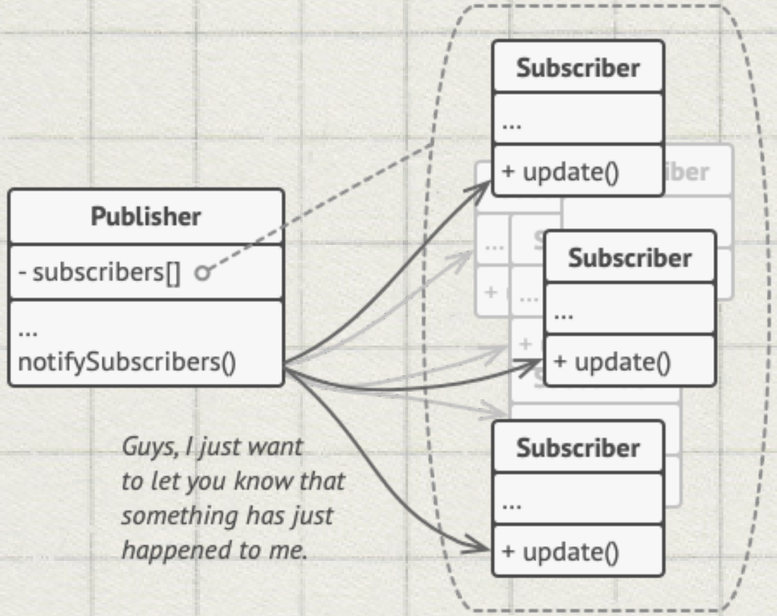
- two types of objects: Customer and Store
- Customer is interested in a particular product which should become available in the Store very soon
- Customer could visit the store daily to check the abvailability, but most of the trips would be pointless
- Store could send emails to all the customer when a new product becomes available, but this might upset other customers that are not interested in new products



Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

SOLUTION

- the object with interesting state - subject/publisher
- the objects that want to track changes to the publisher's state - subscribers
- subscription mechanism for the Publisher class so individual object can subscribe to/unsubscribe from a stream of events

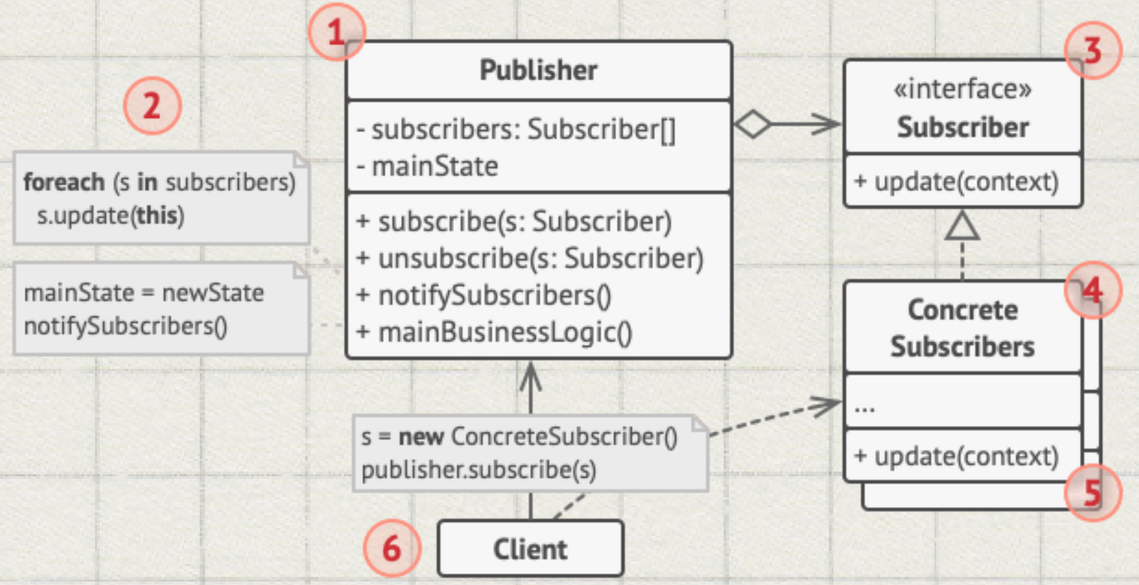


- when an important event happens to the publisher, it notifies each subscriber
- all subscribers implement the same interface and the publisher communicates with them only via that interface

Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

STRUCTURE

- Publisher - issues events of interest when it changes its state or executes some behaviors; has a subscription infrastructure, notifies each subscriber when a new event happens
- Subscriber interface - notification interface with a single update method (can have several parameters for details)
- Concrete Subscribers - implement the interface, perform actions in response to publisher's notifications
- Client - creates publisher and subscriber objects separately and then registers subscribers for publisher updates



Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

- used 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
- usually, this problem comes when working with graphical user interfaces
- used when objects in the app must observe others, but only for a limited time or in specific cases

Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

REAL-WORLD ANALOGY

- subscriptions to newspapers or magazines
- the publisher sends new issues directly to your mailbox right after publication/in advance
- the publisher maintains a list of subscribers and magazines in which they're interested
- subscribers can leave the list at any time if they wish to stop the service

Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

```java
class Publisher {
    3 usages
    private Set<Observer> subscribers = new HashSet<>();

    2 usages
    public void subscribe(Observer observer) {
        subscribers.add(observer);
    }

    1 usage
    public void unsubscribe(Observer observer) {
        subscribers.remove(observer);
    }

    2 usages
    public void notifySubscribers() {
        for (Observer observer : subscribers) {
            observer.update();
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Publisher publisher = new Publisher();
        Subscriber subscriber1 = new Subscriber( name: "Alice");
        Subscriber subscriber2 = new Subscriber( name: "Bob");

        publisher.subscribe(subscriber1);
        publisher.subscribe(subscriber2);

        publisher.notifySubscribers();
        // Output:
        // Alice received an update from the publisher
        // Bob received an update from the publisher

        publisher.unsubscribe(subscriber1);

        publisher.notifySubscribers();
        // Output:
        // Bob received an update from the publisher
    }
}
```

```java
class Subscriber implements Observer {
    2 usages
    private String name;

    2 usages
    public Subscriber(String name) {
        this.name = name;
    }
    1 usage
    public void update() {
        System.out.println(name + " received an update from the publisher");
    }
}
```

```java
interface Observer {
    1 usage  1 implementation
    void update();
}
```

PROS

- ✓ Open/Closed Principle - new subscriber classes can be introduced 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

CONS

- ✗ Subscribers are notified in random order

Source: "Observer". Refactoring.Guru, 2023, https://refactoring.guru/design-patterns/observer. Accessed 26 Apr 2023.

REAL-LIFE USAGE OF OBSERVER PATTERN

The functionalities that the Observer design pattern provides have been used more frequently as technology became a main part of our lives.

Today, most of us benefit from the usage of such a design pattern without even knowing until further analysis. Among the usages of the Observer pattern, we may notice newsletters, notification systems (social media, weather apps, stocks, emergency systems), multiplayer games (synchronization of all player events), etc.

However, the most noticeable feature for us, the developers, is the reactivity that is achieved through this observe-and-act system.

ANGULAR

https://github.com/angular/angular/blob/main/packages/common/http/src/client.ts

https://github.com/ReactiveX/rxjs/blob/master/src/internal/Observable.ts

```
/**
 * Constructs a request that interprets the body as an `ArrayBuffer` and returns the response in
 * an `ArrayBuffer`.
 *
 * @param method  The HTTP method.
 * @param url     The endpoint URL.
 * @param options The HTTP options to send with the request.
 *
 *
 * @return An `Observable` of the response, with the response body as an `ArrayBuffer`.
 */
request(method: string, url: string, options: {
  body?: any,
  headers?: HttpHeaders|{[header: string]: string | string[]},
  context?: HttpContext,
  observe?: 'body',
  params?: HttpParams|
        {[param: string]: string | number | boolean | ReadonlyArray<string|number|boolean>},
  reportProgress?: boolean, responseType: 'arraybuffer',
  withCredentials?: boolean,
}): Observable<ArrayBuffer>;
```

```
it('should have GET response header from test interceptor', waitForAsync(() => {
     let gotResponse = false;
     const req = new HttpRequest<any>('GET', 'api/heroes');
     http.request<Hero[]>(req).subscribe(event => {
       if (event.type === HttpEventType.Response) {
         gotResponse = true;

         const resHeader = event.headers.get('x-test-res');
         expect(resHeader).toBe('res-test-header');

         const heroes = event.body as Hero[];
         expect(heroes.length).toBeGreaterThan(0, 'should have heroes');
       }
     }, failRequest, () => expect(gotResponse).toBe(true, 'should have seen Response event'));
   }));
```

```
subscribe(observerOrNext?: Partial<Observer<T>> | ((value: T) => void) | null): Subscription {
  const subscriber = observerOrNext instanceof Subscriber ? observerOrNext : new Subscriber(observerOrNext);
  subscriber.add(this._trySubscribe(subscriber));
  return subscriber;
}
```

FLUTTER

https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/foundation/change_notifier.dart

```dart
@override
void addListener(VoidCallback listener) {
  assert(ChangeNotifier.debugAssertNotDisposed(this));
  if (kFlutterMemoryAllocationsEnabled && !_creationDispatched) {
    MemoryAllocations.instance.dispatchObjectCreated(
      library: _flutterFoundationLibrary,
      className: '$ChangeNotifier',
      object: this,
    );
    _creationDispatched = true;
  }
  if (_count == _listeners.length) {
    if (_count == 0) {
      _listeners = List<VoidCallback?>.filled(1, null);
    } else {
      final List<VoidCallback?> newListeners =
          List<VoidCallback?>.filled(_listeners.length * 2, null);
      for (int i = 0; i < _count; i++) {
        newListeners[i] = _listeners[i];
      }
      _listeners = newListeners;
    }
  }
  _listeners[_count++] = listener;
}
```

```dart
@override
void removeListener(VoidCallback listener) {
  // This method is allowed to be called on disposed instances for usability
  // reasons. Due to how our frame scheduling logic between render objects and
  // overlays, it is common that the owner of this instance would be disposed a
  // frame earlier than the listeners. Allowing calls to this method after it
  // is disposed makes it easier for listeners to properly clean up.
  for (int i = 0; i < _count; i++) {
    final VoidCallback? listenerAtIndex = _listeners[i];
    if (listenerAtIndex == listener) {
      if (_notificationCallStackDepth > 0) {
        // We don't resize the list during notifyListeners iterations
        // but we set to null, the listeners we want to remove. We will
        // effectively resize the list at the end of all notifyListeners
        // iterations.
        _listeners[i] = null;
        _reentrantlyRemovedListeners++;
      } else {
        // When we are outside the notifyListeners iterations we can
        // effectively shrink the list.
        _removeAt(i);
      }
      break;
    }
  }
}
```

```dart
abstract class Listenable {
  /// Abstract const constructor. This constructor enables subclasses to provide
  /// const constructors so that they can be used in const expressions.
  const Listenable();

  /// Return a [Listenable] that triggers when any of the given [Listenable]s
  /// themselves trigger.
  ///
  /// The list must not be changed after this method has been called. Doing so
  /// will lead to memory leaks or exceptions.
  ///
  /// The list may contain nulls; they are ignored.
  factory Listenable.merge(List<Listenable?> listenables) = _MergingListenable;

  /// Register a closure to be called when the object notifies its listeners.
  void addListener(VoidCallback listener);

  /// Remove a previously registered closure from the list of closures that the
  /// object notifies.
  void removeListener(VoidCallback listener);
}
```

OTHER REAL-LIFE EXAMPLES