Behavioral Design Pattern

Instructor: Mehroze Khan

Behavioral Design Pattern

- Behavioral design patterns are concerned with algorithms and the **assignment of responsibilities** between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the **patterns of communication** between them.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

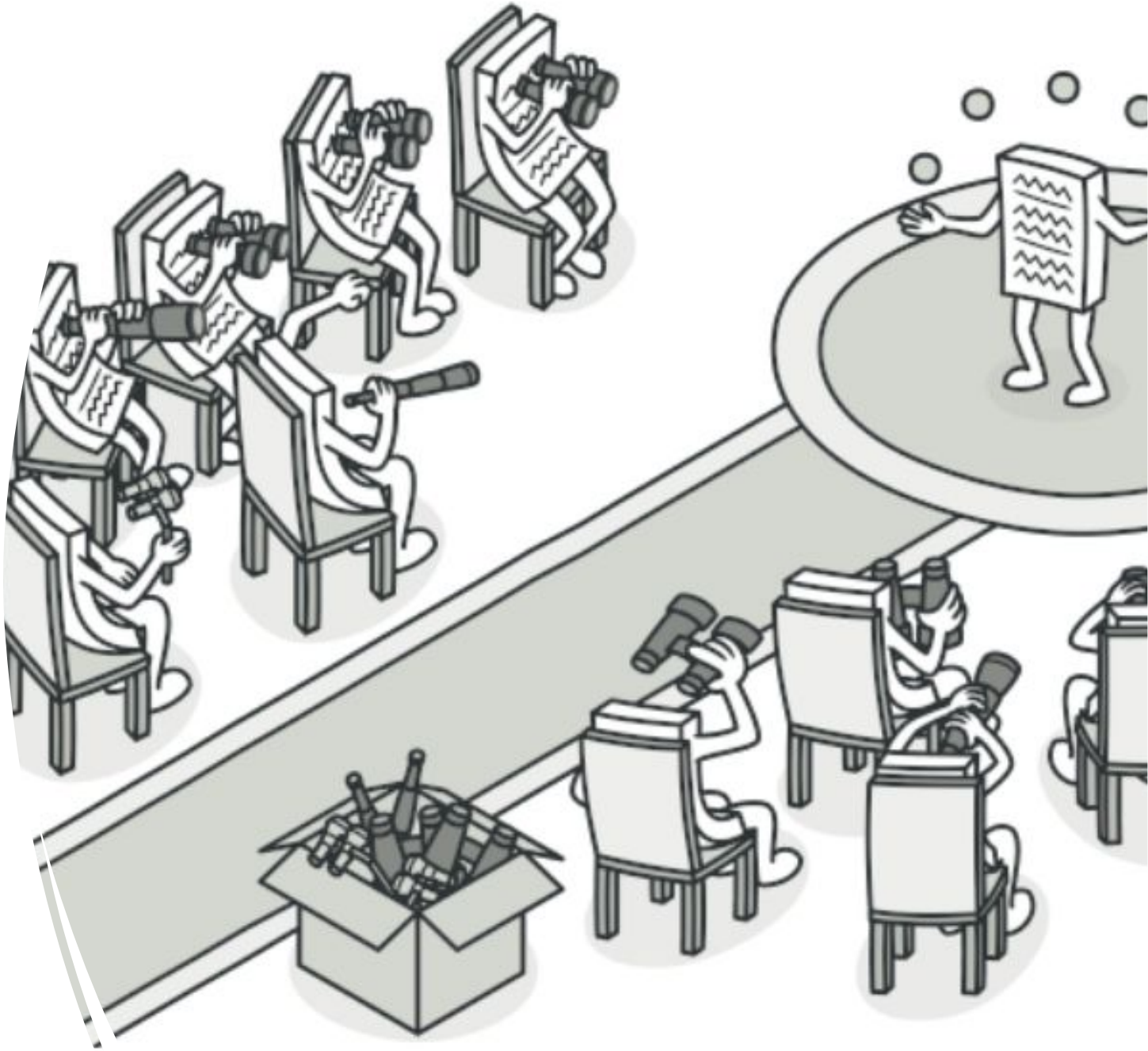
Observer Pattern

Observer Pattern

Also Known as:
Event-Subscriber, Listener

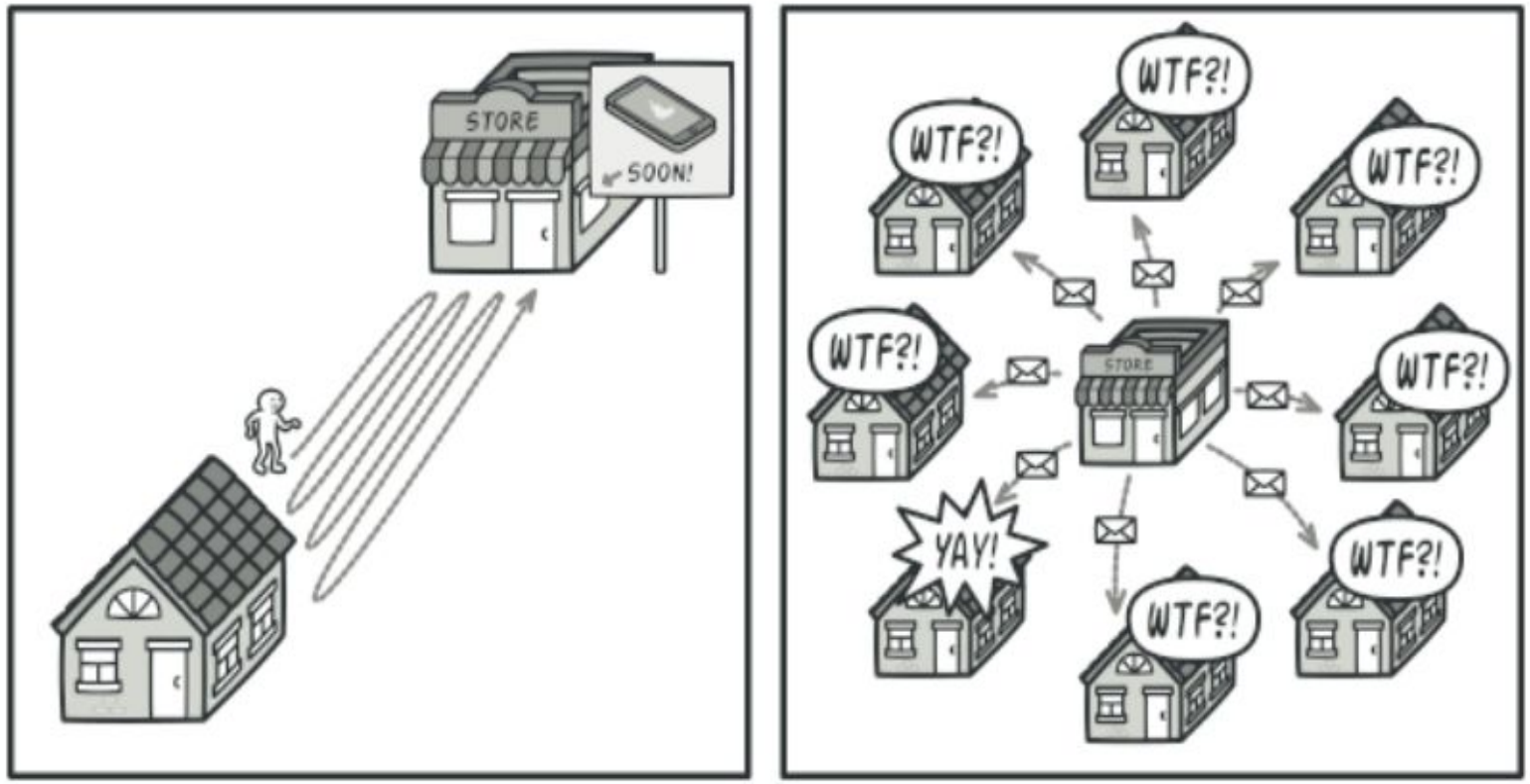
Intent:

- **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

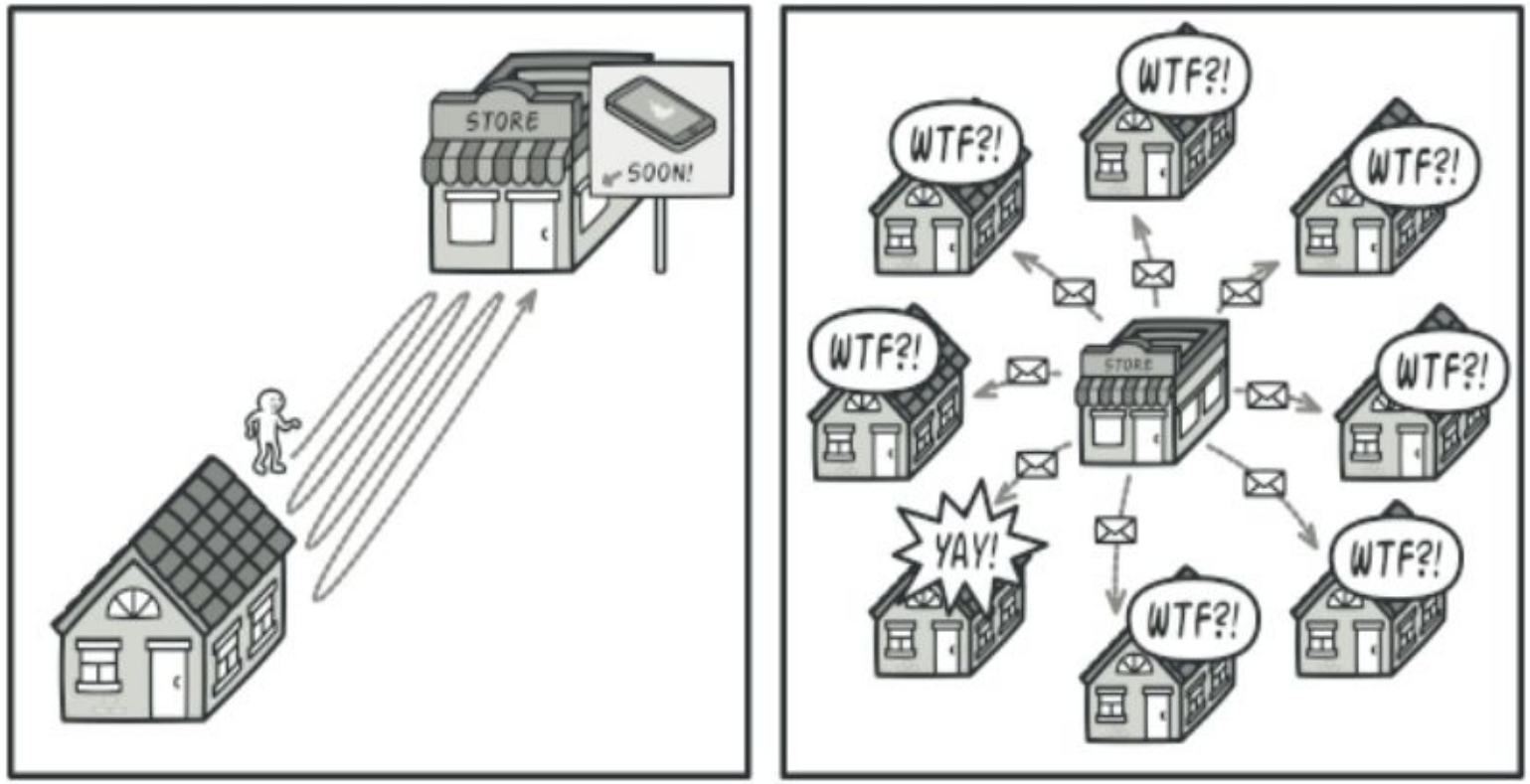
- Imagine that you have two types of objects: a **Customer** and a **Store**.
- The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.
- The customer could visit the store every day and check product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



Visiting the store vs. sending spam

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.



Visiting the store vs. sending spam

Solution

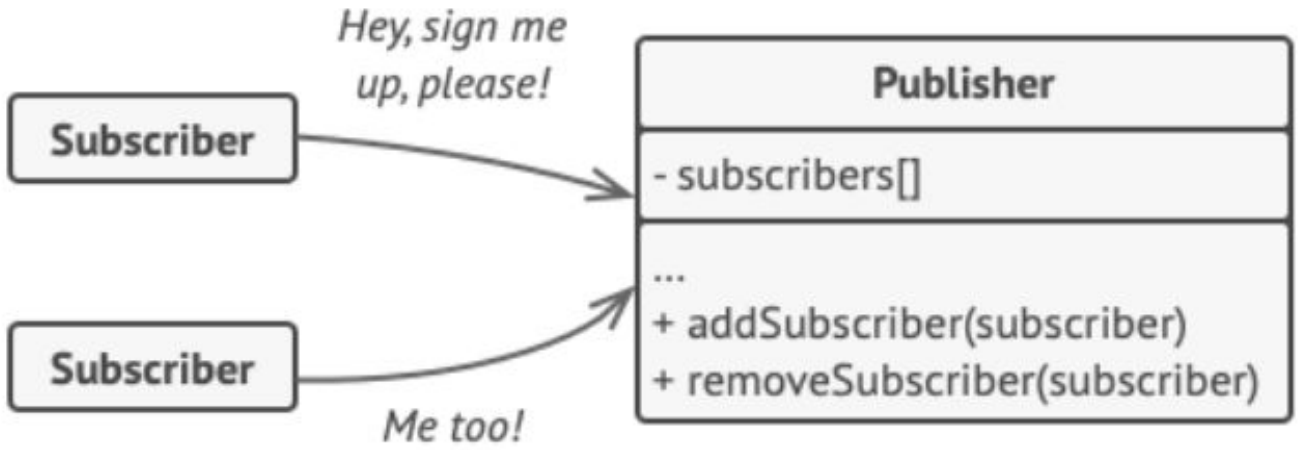
- The object that has some interesting state is often called ***subject***, but since it's also going to notify other objects about the changes to its state, we'll call it ***publisher***.
- All other objects that want to track changes to the publisher's state are called ***subscribers***.

Solution

- 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.
- This mechanism consists of:

  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.



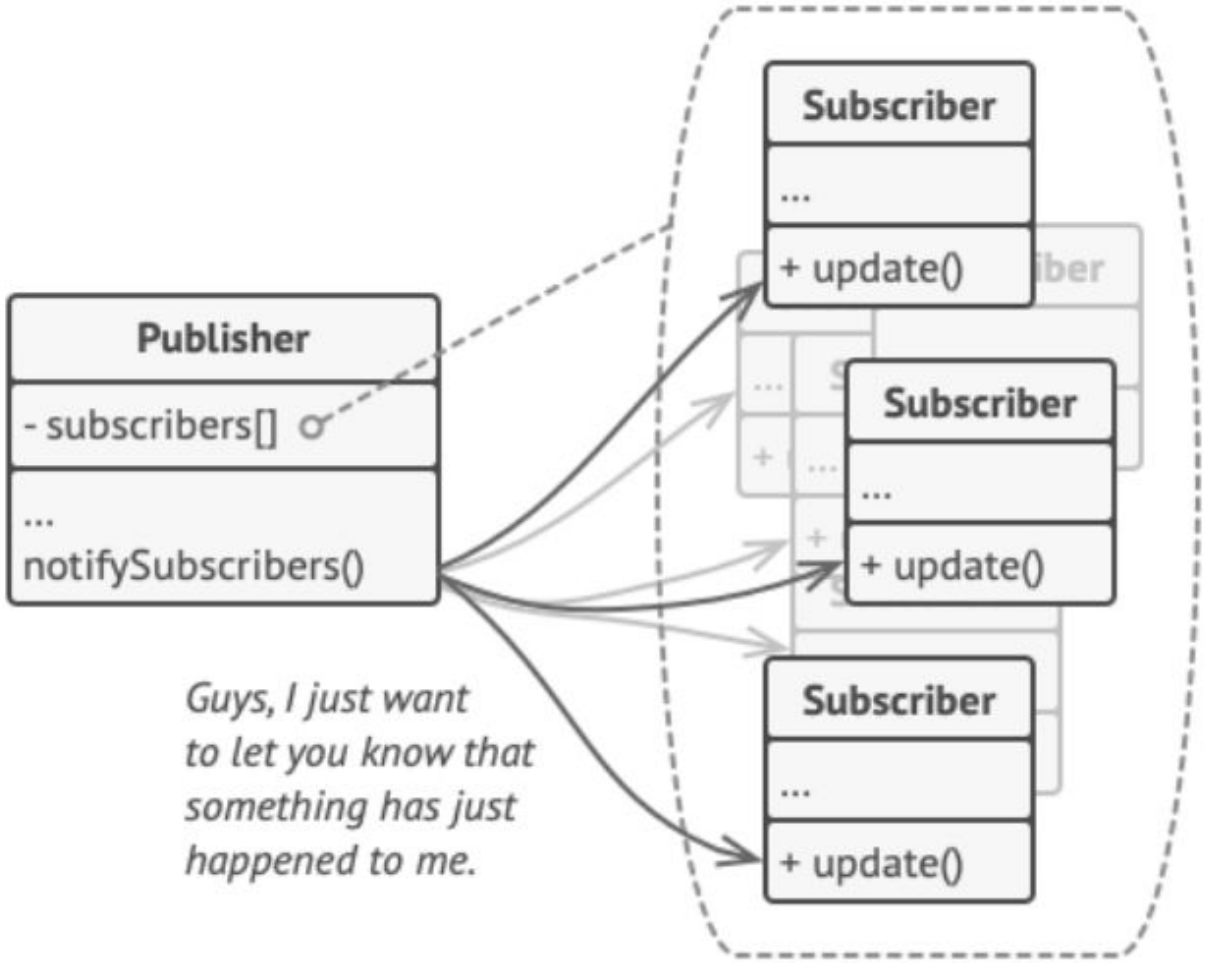
A subscription mechanism lets individual objects subscribe to event notifications.

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 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.

Solution

- That's why 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.
- 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 few subscription methods. The interface would allow subscribers to observe publishers' states without coupling to their concrete classes.

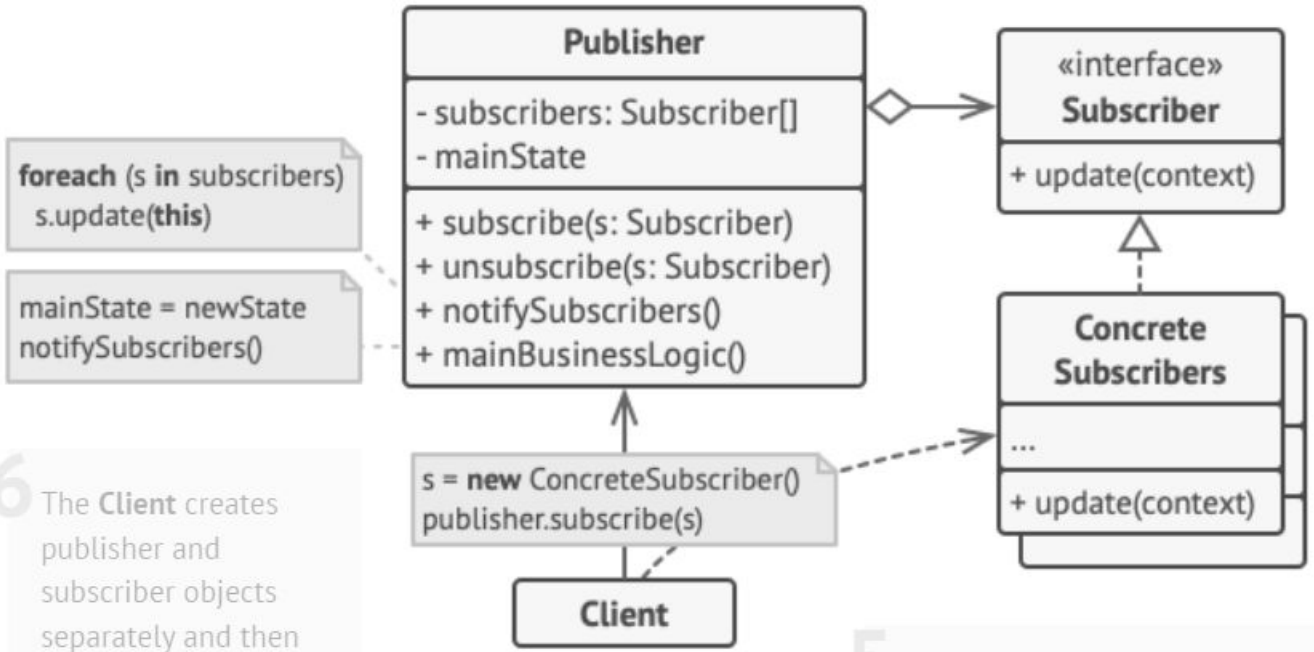


Publisher notifies subscribers by calling the specific notification method on their objects.

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

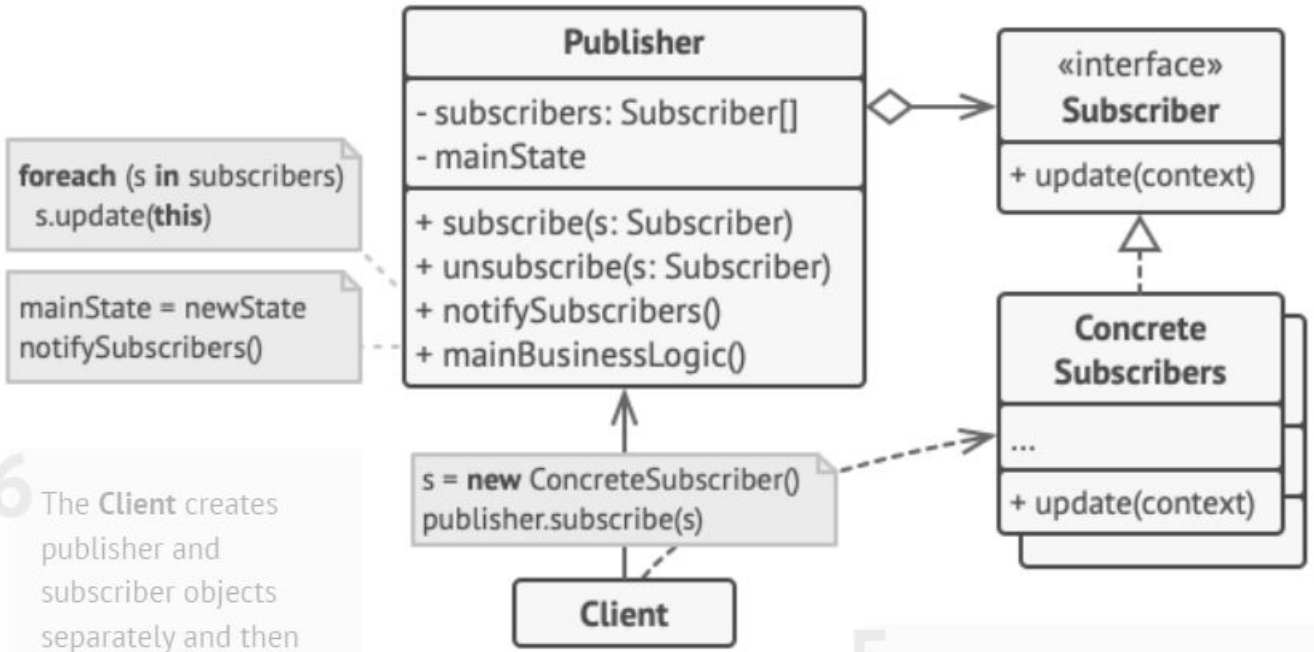
5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

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

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

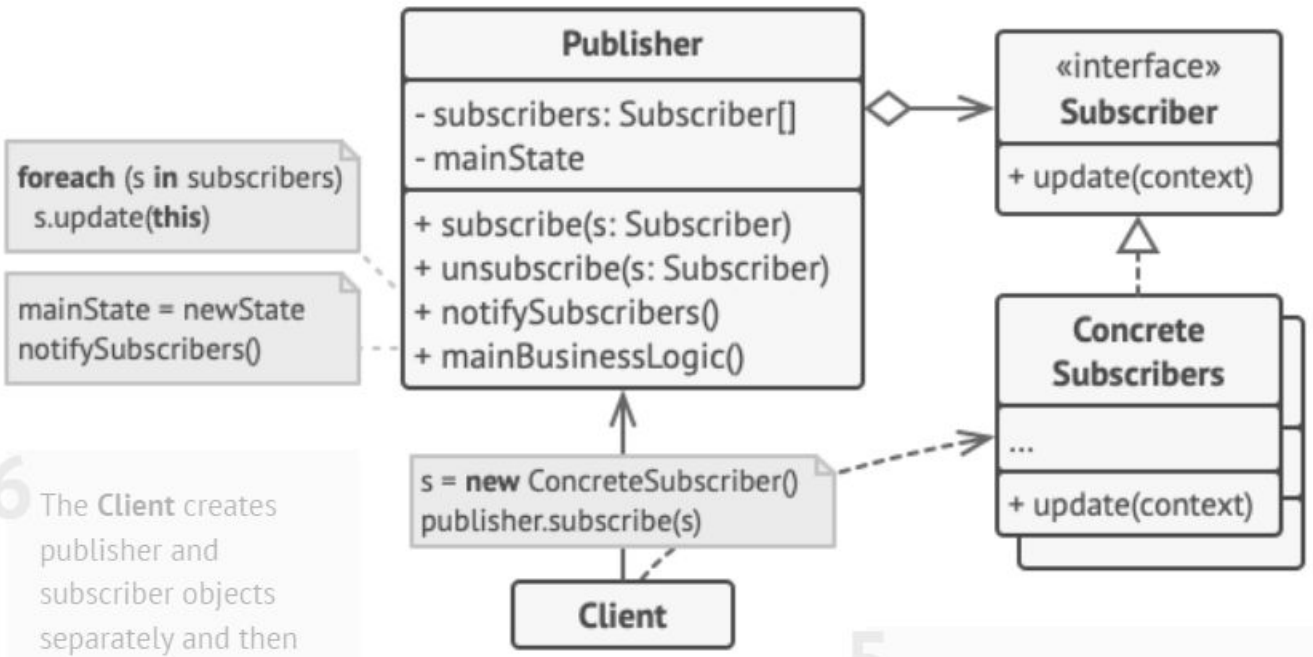
5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

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

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

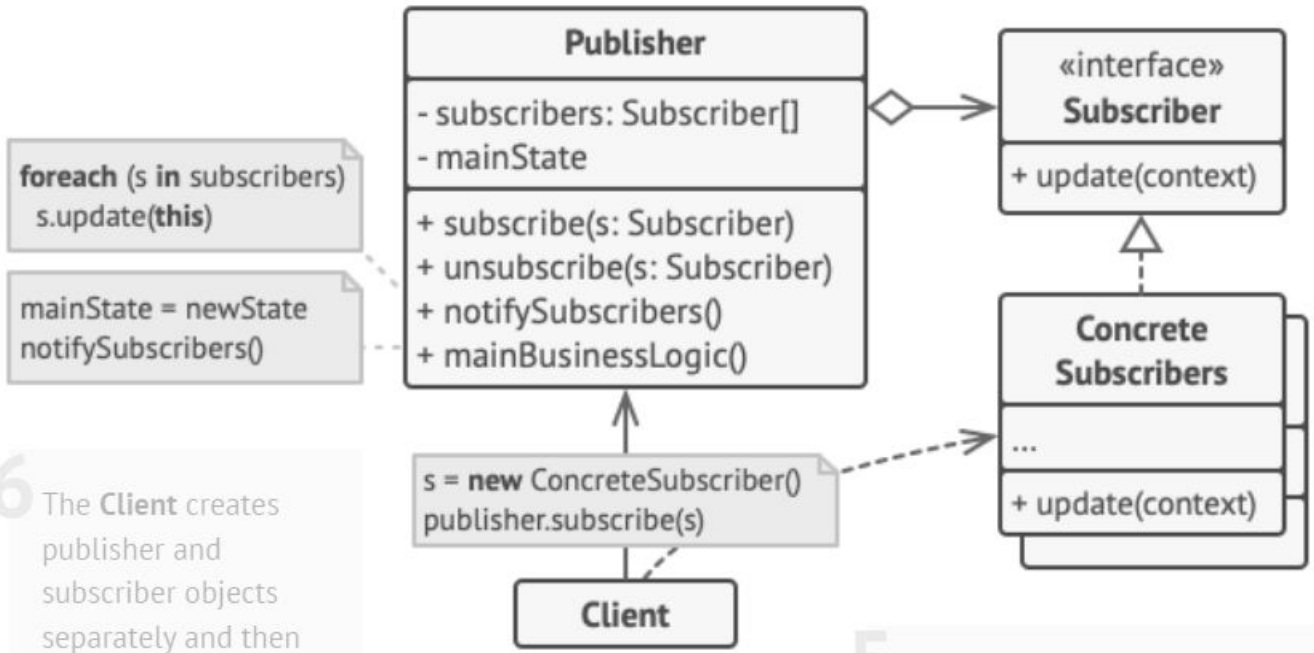
5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

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

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

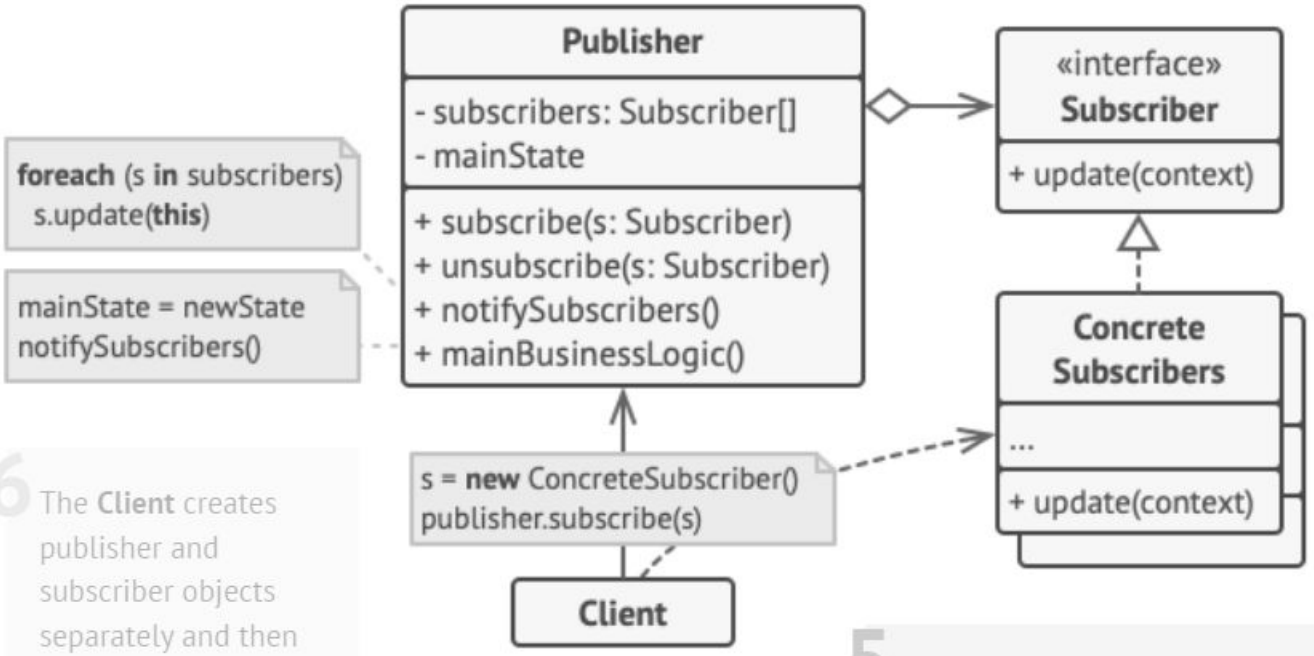
5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

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

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

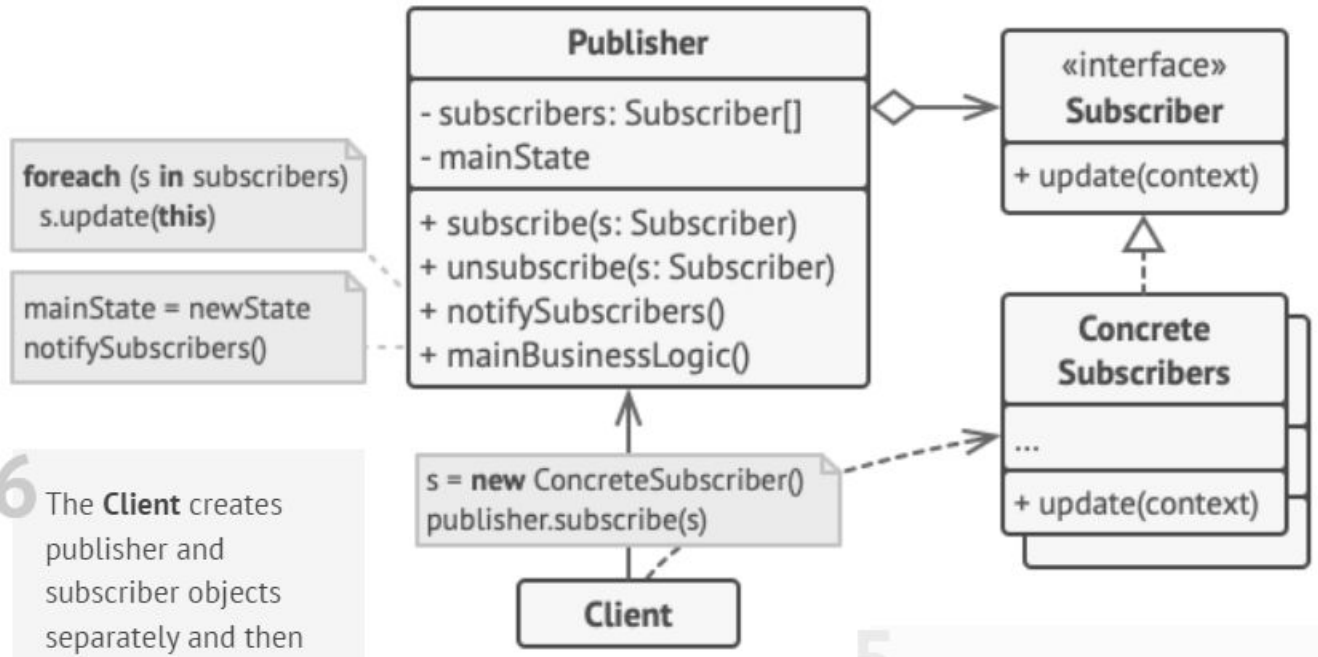
5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

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

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 infrastructure that lets new subscribers join and current subscribers leave the list.

2 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.



3 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.

4 **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.

5 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.

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

Example

```
// Abstract Subscriber class
class Subscriber {
public:
    virtual void update(const string& message) = 0;
};

// Concrete Subscriber class
class ConcreteSubscriber : public Subscriber {
public:
    void update(const string& message) override {
        cout << "Received update: " << message << endl;
    }
};
```

```
// Publisher class
class Publisher {
private:
    list<Subscriber*> subscribers;
public:
    void subscribe(Subscriber* subscriber) {
        subscribers.push_back(subscriber);
    }
    void unsubscribe(Subscriber* subscriber) {
        subscribers.remove(subscriber);
    }
    void notifySubscribers(const string& message) {
        for (Subscriber* subscriber : subscribers) {
            subscriber->update(message);
        }
    }
};
```

Example

```
// Client
int main() {
    Publisher publisher;
    ConcreteSubscriber subscriber1, subscriber2, subscriber3;

    publisher.subscribe(&subscriber1);
    publisher.subscribe(&subscriber2);
    publisher.subscribe(&subscriber3);

    publisher.notifySubscribers("Data updated!");

    publisher.unsubscribe(&subscriber1);

    publisher.notifySubscribers("Another update!");

    return 0;
}
```

```
// Output
Received update: Data updated!
Received update: Data updated!
Received update: Data updated!

Received update: Another update!
Received update: Another update!
```

Pros and Cons

Pros:

- *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.

Cons:

- Subscribers are notified in random order.

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.

Helping Links:

- https://refactoring.guru/design-patterns/