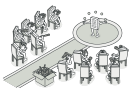
Hey, have you noticed the price for pro-powers, ref.guru/pro-sku-programming-offer-50-off-sep-2022 ? Grab it before it is gone!

Observer

Also known as: Event Subscriber, Listener

Intent

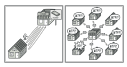
Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Problem

Imagine that you have two new 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 on mark, most of these trips would be pointless.



Visiting the store vs. notifying clients

On the other hand, the store could send tons of emails (which might be annoyed queries as all customers want to be a new product becomes available. This would save some customers from endless trips to the store. At the same time, it's a pain also a customer who wasn't interested in new products.

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

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.

Publisher notifies interested objects about events that occur in its object's membership.

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 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.

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

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 mention how the subscribe mechanism. The interface would allow subscribers to observe publishers' states without coupling to their concrete classes.

Real-World Analogy

Magazine subscription.

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 (after either 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 anytime when they wish to stop the publisher sending new magazine issues to them.

Structure

1. The **Publisher** issues events of interests 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 any 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 subscribers fetch any required data directly.

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

Pseudocode

In this example, the **Observer** pattern lets the text editor object notify other service objects about changes to its state.

Anything happening inside the editor triggers events. Other classes can listen to these events and follow the editor's updates.

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 dedicated to just that. You could upgrade that object to serve as a centralized event dispatcher, letting any object act as a publisher.

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.

```
// The basic publisher class includes subscription management
// and one utility method for updates.
class EventManager {
    private protected function: array listeners = array(), function
    function __construct() {
        this->listeners = array();
    }

    function subscribe(eventType, listener) {
        this->listeners[eventType][] = listener;
    }

    function unsubscribe(eventType, listener) {
        unset(this->listeners[eventType][array_search(listener,
            this->listeners[eventType], true)]);
    }

    function notify(eventType, data = null) {
        foreach (this->listeners[eventType] as listener) {
            listener.update(data);
        }
    }
}

// The concrete publisher extends base business logic class,
// inheriting the base subscription. We could write this class
// from scratch, but in real life we'd rather use existing code
// when it's tested. The concrete publisher might already be a
// descendant of some other business class, in this case we'll
// use composition, as we did here.
class Editor {
    public events: EventManager
    private file: File = null
    private content: String = null

    constructor Editor() {
        this->events = new EventManager();
    }

    // Methods of business logic can notify subscribers about
    // important events.
    function openFile(path) {
        this->file = new File(path);
        this->content = this->file->read();
        this->events->notify("open", this->file.name);
    }

    function saveFile() {
        this->file->write(this->content);
        this->events->notify("save", this->file.name);
    }

    // ...

}

// Easy to add subscriber interfaces. If your programming language
// supports functional types, you can replace the whole
// subscriber hierarchy with a set of functions.
interface SubscribeEvents {
    function update(filename);
}

// Concrete subscribers react to updates issued by the publisher they
// are attached to.
class LoggingListener implements SubscribeEvents {
    private log: File = null;
    private function message(msg: string)

    constructor LoggingListener(log_filename, message) {
        this->log = new File(log_filename);
        this->message = message;
    }

    function update(filename) {
        this->log->write(replace('%s', filename, this->message));
    }
}

class EmailNotificationListener implements SubscribeEvents {
    private email: string;
    private function message(msg: string)

    constructor EmailNotificationListener(email, message) {
        this->email = email;
        this->message = message;
    }

    function update(filename) {
        system.mail(this->email, 'Notification', 'Someone has opened file: '
            . filename . ' at ' . time());
    }
}

// The application can configure publishers and subscribers at
// runtime.
class ApplicationConfig {
    function setup() {
        editor = new Editor();

        logger = new LoggingListener(
            '/path/to/log.txt',
            'Someone has opened file: %s at ' . time()
        );
        editor.events.subscribe('open', logger);

        emailListener = new EmailNotificationListener(
            'admin@example.com',
            'Someone has changed the file: %s'
        );
        editor.events.subscribe('save', emailListener);
```

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.

You can often encounter this problem when working with classes of the graphical user interface. For example, you created custom button classes, and you want to let the client code know custom code ran in your button(s) in that this unknown to user processes or modes.

The Observer pattern lets any object that implements the subscriber interface subscribe to some notifications in publisher objects. You can add the subscription mechanism to your buttons, letting the clients hook up their custom code via custom subscriber classes.

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

The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.

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 serve 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.

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.

   However, if you're applying the pattern to an existing class hierarchy, consider an approach based on composition: put the subscription logic into a separate object, and make all real publishers use it.

5. Create concrete publisher classes. Each time something important happens inside a publisher, it must notify all its subscribers.

6. Implement the update notification methods in concrete subscriber classes. Most subscribers 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.

Pros and Cons

✔ Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher's code (and code base of other publisher classes).

✔ You can establish relations between objects at runtime.

✘ Subscribers are notified in random order.

✔ You can establish relations between objects at runtime.

Relations with Other Patterns

- Chain of Responsibility, Command, Mediator, and Observer address various ways of connecting senders and receivers of requests:
  - Chain of Responsibility passes a request sequentially along a dynamic chain of potential receivers until one of them handles it.
  - Command establishes unidirectional connections between senders and receivers.
  - Mediator eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.
  - Observer lets receivers dynamically subscribe to and unsubscribe from receiving requests.
- The difference between **Mediator** and **Observer** is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.

  The primary goal of Mediator is to eliminate mutual dependencies among a set of system components. Instead, those components become dependent on a single mediator class. The goal of Observer is to establish dynamic one-way connections between objects, where some objects act as subordinates to others.

  There's a popular implementation of the Mediator pattern that relies on Observer. The mediator object plays the role of publisher, and the components act as subscribers which subscribe to and unsubscribe from the mediator's events. When Mediator is implemented this way, it may look very similar to Observer.

  When you're confused, remember that you can implement the Mediator pattern in other ways. For example, you can permanently link all dialog components to the chief mediator object. This implementation won't resemble Observer but still falls as an instance of the Mediator pattern.

  Now imagine a program where all components have become publishers, allowing dynamic connections between each other. There won't be a centralized mediator object, only a distributed web of listeners.

Code Examples

Support our free website and own the eBook!

- 1000+ diagrams and illustrations expressed in 3 languages
- PDF and Kindle ebook, easy to read, print or take along
- Ad-free experience, no cookies, no third-party ads
- All updates with code examples, in 8 languages
- We donate a portion of our proceeds to charities

Learn more

© 2014-2024 Refactoring.Guru. All rights reserved.
Illustrations by Dmitry Zhart
Refactoring.Guru, Inc. 171 Sunnyside Blvd#B, Plainview, 11803
Email: support@refactoring.guru
Terms & Conditions
Privacy Policy
Cookie Usage Policy