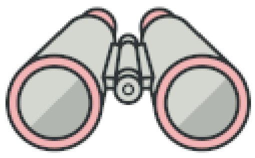

Observer en PHP

Observer es un patrón de diseño de comportamiento que permite a un objeto notificar a otros objetos sobre cambios en su estado.

El patrón Observer proporciona una forma de suscribirse y cancelar la subscripción a estos eventos para cualquier objeto que implementa una interfaz suscriptora.

Aprende más sobre el patrón Observer →

Navegación

Complejidad: ★★☆

Popularidad: ★★★

Ejemplos de uso: PHP tiene varias interfaces integradas (**SplSubject**, **SplObserver**) que pueden utilizarse para hacer tus implementaciones del patrón Observer compatibles con el resto del código PHP.

objetos en una lista, y por las llamadas al método de actualización emitidas a todos los objetos de esa lista.

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Observer** y se centra en las siguientes preguntas:

- ¿De qué clases se compone?
- ¿Qué papeles juegan esas clases?
- ¿De qué forma se relacionan los elementos del patrón?

Después de conocer la estructura del patrón, será más fácil comprender el siguiente ejemplo basado en un caso de uso real de PHP.

index.php: Ejemplo conceptual

```
<?php

namespace RefactoringGuru\Observer\Conceptual;

/**
 * PHP has a couple of built-in interfaces related to the Observer pattern.
 *
 * Here's what the Subject interface looks like:
 *
 * @link http://php.net/manual/en/class.splsubject.php
 *
 *     interface SplSubject
 *     {
 *         // Attach an observer to the subject.
 *         public function attach(SplObserver $observer);
 *
 *         // Detach an observer from the subject.
 *         public function detach(SplObserver $observer);
 *
 *         // Notify all observers about an event.
 *         public function notify();
 *     }
 *
 * There's also a built-in interface for Observers:
 *
 * @link http://php.net/manual/en/class.splobserver.php
```

```php
 *     {
 *         public function update(SplSubject $subject);
 *     }
 */

/**
 * The Subject owns some important state and notifies observers when the state
 * changes.
 */
class Subject implements \SplSubject
{
    /**
     * @var int For the sake of simplicity, the Subject's state, essential to
     * all subscribers, is stored in this variable.
     */
    public $state;

    /**
     * @var \SplObjectStorage List of subscribers. In real life, the list of
     * subscribers can be stored more comprehensively (categorized by event
     * type, etc.).
     */
    private $observers;

    public function __construct()
    {
        $this->observers = new \SplObjectStorage();
    }

    /**
     * The subscription management methods.
     */
    public function attach(\SplObserver $observer): void
    {
        echo "Subject: Attached an observer.\n";
        $this->observers->attach($observer);
    }

    public function detach(\SplObserver $observer): void
    {
        $this->observers->detach($observer);
        echo "Subject: Detached an observer.\n";
    }

    /**
     * Trigger an update in each subscriber.
     */
    public function notify(): void
    {
        echo "Subject: Notifying observers...\n";
        }
    }

    /**
     * Usually, the subscription logic is only a fraction of what a Subject can
     * really do. Subjects commonly hold some important business logic, that
     * triggers a notification method whenever something important is about to
     * happen (or after it).
     */
    public function someBusinessLogic(): void
    {
        echo "\nSubject: I'm doing something important.\n";
        $this->state = rand(0, 10);

        echo "Subject: My state has just changed to: {$this->state}\n";
        $this->notify();
    }
}

/**
 * Concrete Observers react to the updates issued by the Subject they had been
 * attached to.
 */
class ConcreteObserverA implements \SplObserver
{
    public function update(\SplSubject $subject): void
    {
        if ($subject->state < 3) {
            echo "ConcreteObserverA: Reacted to the event.\n";
        }
    }
}

class ConcreteObserverB implements \SplObserver
{
    public function update(\SplSubject $subject): void
    {
        if ($subject->state == 0 || $subject->state >= 2) {
            echo "ConcreteObserverB: Reacted to the event.\n";
        }
    }
}

/**
 * The client code.
 */

$subject = new Subject();

$o1 = new ConcreteObserverA();
```

```
$o2 = new ConcreteObserverB();
$subject->attach($o2);

$subject->someBusinessLogic();
$subject->someBusinessLogic();

$subject->detach($o2);

$subject->someBusinessLogic();
```

Output.txt: Resultado de la ejecución

```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 2
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 4
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 1
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
```

Ejemplo del mundo real

En este ejemplo, el patrón **Observer** permite a varios objetos observar eventos que suceden dentro del repositorio de usuario de una aplicación.

El repositorio emite varios tipos de eventos y permite a los observadores escucharlos todos, o solo algunos.

index.php: Ejemplo del mundo real

```
namespace RefactoringGuru\Observer\RealWorld;

/**
 * The UserRepository represents a Subject. Various objects are interested in
 * tracking its internal state, whether it's adding a new user or removing one.
 */
class UserRepository implements \SplSubject
{
    /**
     * @var array The list of users.
     */
    private $users = [];

    // Here goes the actual Observer management infrastructure. Note that it's
    // not everything that our class is responsible for. Its primary business
    // logic is listed below these methods.

    /**
     * @var array
     */
    private $observers = [];

    public function __construct()
    {
        // A special event group for observers that want to listen to all
        // events.
        $this->observers["*"] = [];
    }

    private function initEventGroup(string $event = "*"): void
    {
        if (!isset($this->observers[$event])) {
            $this->observers[$event] = [];
        }
    }

    private function getEventObservers(string $event = "*"): array
    {
        $this->initEventGroup($event);
        $group = $this->observers[$event];
        $all = $this->observers["*"];

        return array_merge($group, $all);
    }

    public function attach(\SplObserver $observer, string $event = "*"): void
    {
        $this->initEventGroup($event);
```

```php
public function detach(\SplObserver $observer, string $event = "*"): void
{
    foreach ($this->getEventObservers($event) as $key => $s) {
        if ($s === $observer) {
            unset($this->observers[$event][$key]);
        }
    }
}

public function notify(string $event = "*", $data = null): void
{
    echo "UserRepository: Broadcasting the '$event' event.\n";
    foreach ($this->getEventObservers($event) as $observer) {
        $observer->update($this, $event, $data);
    }
}

// Here are the methods representing the business logic of the class.

public function initialize($filename): void
{
    echo "UserRepository: Loading user records from a file.\n";
    // ...
    $this->notify("users:init", $filename);
}

public function createUser(array $data): User
{
    echo "UserRepository: Creating a user.\n";

    $user = new User();
    $user->update($data);

    $id = bin2hex(openssl_random_pseudo_bytes(16));
    $user->update(["id" => $id]);
    $this->users[$id] = $user;

    $this->notify("users:created", $user);

    return $user;
}

public function updateUser(User $user, array $data): User
{
    echo "UserRepository: Updating a user.\n";

    $id = $user->attributes["id"];
    if (!isset($this->users[$id])) {
        return null;
        $user = $this->users[$id];
        $user->update($data);

        $this->notify("users:updated", $user);

        return $user;
    }

    public function deleteUser(User $user): void
    {
        echo "UserRepository: Deleting a user.\n";

        $id = $user->attributes["id"];
        if (!isset($this->users[$id])) {
            return;
        }

        unset($this->users[$id]);

        $this->notify("users:deleted", $user);
    }
}

/**
 * Let's keep the User class trivial since it's not the focus of our example.
 */
class User
{
    public $attributes = [];

    public function update($data): void
    {
        $this->attributes = array_merge($this->attributes, $data);
    }
}

/**
 * This Concrete Component logs any events it's subscribed to.
 */
class Logger implements \SplObserver
{
    private $filename;

    public function __construct($filename)
    {
        $this->filename = $filename;
        if (file_exists($this->filename)) {
            unlink($this->filename);
        }
    }
    {
        $entry = date("Y-m-d H:i:s") . ": '$event' with data '" . json_encode($data) . "'
        file_put_contents($this->filename, $entry, FILE_APPEND);

        echo "Logger: I've written '$event' entry to the log.\n";
    }
}

/**
 * This Concrete Component sends initial instructions to new users. The client
 * is responsible for attaching this component to a proper user creation event.
 */
class OnboardingNotification implements \SplObserver
{
    private $adminEmail;

    public function __construct($adminEmail)
    {
        $this->adminEmail = $adminEmail;
    }

    public function update(\SplSubject $repository, string $event = null, $data = null):
    {
        // mail($this->adminEmail,
        //     "Onboarding required",
        //     "We have a new user. Here's his info: " .json_encode($data));

        echo "OnboardingNotification: The notification has been emailed!\n";
    }
}

/**
 * The client code.
 */

$repository = new UserRepository();
$repository->attach(new Logger(__DIR__ . "/log.txt"), "*");
$repository->attach(new OnboardingNotification("1@example.com"), "users:created");

$repository->initialize(__DIR__ . "/users.csv");

// ...

$user = $repository->createUser([
    "name" => "John Smith",
    "email" => "john99@example.com",
]);

// ...
```

Output.txt: Resultado de la ejecución

Ukrainian office:

FOP Olga Skobeleva

Abolmasova 7

Kyiv, Ukraine, 02002

Email: support@refactoring.guru

Spanish office:

Oleksandr Shvets

Avda Pamplona 63, 4b

Pamplona, Spain, 31010

Email: spain@refactoring.guru