Obserwator

Jeśli programujesz w dowolnym frameworku, to z pewnością o nim słyszałeś. Obserwator. Czy to dobry wzorzec? Tak, ale jego użycie niekoniecznie zawsze musi być najlepsze i o tym też opowiem Ci w dzisiejszej lekcji.

Obserwator pozwala zdefiniować mechanizm subskrypcji w celu powiadomienia wielu obiektów o zdarzeniach dziejących się w obserwowanym obiekcie.

Z jednej strony świetnie mieć takie rozwiązanie. Z drugiej jednak strony opieranie na tym kompletnej logiki działania aplikacji, może być bardzo dużym błędem.

Czemu zaczynam od przestrogi? Bo observer bardzo często jest nadużywany, na przykład w Laravelu. Zapisujesz coś do bazy i odpalasz observatora. Czy napewno powinno tak być? Być może wcale nie chcesz zaobserwować zapisania określonej wartości czy jej zmiany w bazie? Być może lepiej byłoby zaobserwować jej faktyczną akcję.

Weźmy dla przykładu sytuację w której po utworzeniu nowego zamówienia, chcielibyśmy zarezerwować wysyłkę. W tym celu za każdym razem, gdy nasz framework zapisuje zmianę w zamówieniu odpalamy API kuriera i informujemy go o konieczności odbioru nowej przesyłki.

Nadchodzi jednak ten dzień. Mamy błąd w aplikacji. Trzeba go poprawić. Wchodzisz więc w tinkera i po prostu zmieniasz nieprawidłowe dane. Niby nic prawda?

No - nie do końca, bo właśnie wywołałeś API kuriera, a przecież wcale tego nie chciałeś!

Dlatego właśnie, choć Obserwator to świetny wzorzec musisz mieć na uwadze co tak naprawdę chcesz obserwować. W większości wypadków nie będzie to “zapis do bazy danych” lecz faktyczna zmiana określonej encji. Często też zamiast samego observera lepiej byłoby skorzystać po prostu z wydarzeń.

Mamy to? Ok! To teraz poznajmy observatora nieco bliżej, od tej właściwej strony!

Trzeba też zauważyć, że sam wzorzec jest tak bardzo powszechny w PHP że sam język udostępnia wbudowane interfejsy do jego obsługi.

Obserwator składa się z obiektu publikującego (publishera) i subskrybentów. Za każdym razem, gdy w obiekcie publishera wydarzy się coś istotnego, może on poinformować wszystkich subskrybentów o tym fakcie, dzięki czemu będą oni mogli na niego wpłynąć.

Każdy subskrybent, niezależnie od tego czym konkretnie się zajmuje, musi implementować interfejs odpowiadający subskrybcji publikującego. Dzięki temu ilość subskrybentów może być dowolnie duża, a sam publisher świetnie sobie z tym poradzi!

Może to być również szczególnie przydatne gdy ilość obiektów obserwowanych i obserujących może być duża i chciałbyś by można było ją dowolnie mieszać.

Ok - teoria zanana? Czas przejść do faktycznej implementacji! Oczywiście wykorzystamy tutaj wbudowane w PHP interfejsy `SplSubject` oraz `SplObserver`.

A nasze zadanie? Tworzymy prosty system do tworzenia treści za pośrednictwem markdown. Po jej dodaniu chcemy by można było ją opublikować w serwisach społecznościowych.

Serwisów społecznościowych jest całkiem sporo, dlatego sprzęganie bezpośrednio klas do publikowania z klasą zawierającą treść, nie jest najlepszym pomysłem.

```
class MarkdownContentCreator implements SplSubject {
    private SplObjectStorage $_observers;
    private bool $published = false;

    public function __construct(private string $name, private
        $this->_observers = new SplObjectStorage();
    }

    public function getName(): string {
        return $this->name;
    }

    public function getContent(): string {
        return $this->content;
    }

    public function markdownToHtml() {
```

```
        // Zastąpienie nagłówków
        $this->content = preg_replace('/^# (.*)$/m', '<h1>$1<
        $this->content = preg_replace('/^## (.*)$/m', '<h2>$1
        $this->content = preg_replace('/^### (.*)$/m', '<h3>$
        // Zastąpienie tekstu pogrubionego
        $this->content = preg_replace('/\*\*(.*?)\*\*/', '<st
        // Zastąpienie tekstu pochylonego
        $this->content = preg_replace('/\*(.*?)\*/', '<em>$1<
        // Zastąpienie list
        $this->content = preg_replace('/^\d\. (.*)$/m', '<li>
        $this->content = preg_replace('/<li>(.*?)<\/li>/', '<
        // Zastąpienie kodu
        $this->content = preg_replace('/```(.+?)```/s', '<cod
        // Zastąpienie linków
        $this->content = preg_replace('/\[([^\[]+)\]\(([^)]+)
    }

    public function publish(): void
    {
        $this->markdownToHtml();
        echo 'Treść ' . substr($this->content, 0, 15) . ' zos
        $this->published = true;
    }

    public function attach(SplObserver $observer): void
    {
        $this->_observers->attach($observer);
    }

    public function detach(SplObserver $observer): void
    {
        $this->_observers->detach($observer);
    }

    public function notify(): void
    {
        if(!$this->published) {
            return;
```

```
        }

        foreach ($this->_observers as $observer) {
            $observer->update($this);
        }
    }
}
```

Aż do metody attach nasza klasa to całkowicie normalna kasa, realizująca swoje zadania. Dopiero metody attach, detach i notify realizują określone w interfejsie `SplSubject` zadania.

Metoda attach dołącza poszczególne, nowe klasy obserwatorów.

Detach robi działanie odwrotne, z kolei notify wysyła powiadomienie o aktualizacji, jeśli oczywiście treść została opublikowana.

Skoro to jest jasne, to czas zaprogramować klasy Obserwatorów

```
class FacebookPublisherObserver implements SplObserver {
    public function update(SplSubject $subject): void {
        echo 'Dodaję post na facebooka o nazwie ' . $subject-
    }
}

class InstagramPublisherObserver implements SplObserver {
    public function update(SplSubject $subject): void {
        echo 'Dodaję zdjęcie na instagrama o nazwie ' . $subj
    }
}
```

Są dość podobne, a ich bezpośrednia faktyczna logika będzie dziać się już bezpośrednio w ich środku.

Proste prawda? Teraz wypadałoby wykonać ten kod

```
<?php
$publisher = new MarkdownContentCreator("Tytuł", "## Nagłówek

$observer1 = new FacebookPublisherObserver();
$observer2 = new InstagramPublisherObserver();
```

```
$publisher->attach($observer1);
$publisher->attach($observer2);
$publisher->publish();
$publisher->notify();
```

Tworzymy treść, oraz obserwatorów. Dodajemy ich do naszej listy, a następnie ją publikujemy. Po publikacji zarówno Facebook jak i Instagram usłyszą od publikatora że treść została opublikowana, dzięki wykonaniu metody `notify`.

Zauważ, że metoda ta jest użyta "celowo". Co prawda np. Eloquent pozwala na silent save, ale tutaj możemy faktycznie zdecydować, czy chcemy, czy nie chcemy informować o zmianach.

Wzorzec ten, jak widzisz, pozwala nam również zdecydować kiedy faktycznie chcemy skorzystać z określonych obserwatorów. Możemy je w dowolnym miejscu podczepić i odczepić, jeśli jest to nam potrzebne. Możemy to również uzależnić od określonej logiki, dostępnej jednak poza samą implementacją. Nie tworzy się przez to niepotrzebny coupling i nasz kod wygląda naprawdę dobrze!

Myślę że teraz nie tylko wiesz jak korzystać z tego wzorca, ale wiesz również, jak robić to świadomie.

Wady i zalety

Zalety:

- *Zasada otwarte/zamknięte*. Można wprowadzać do programu nowe klasy subskrybujące bez konieczności zmieniania kodu publikującego (i odwrotnie, jeśli istnieje interfejs publikujący).
- Można utworzyć związek pomiędzy obiektami w trakcie działania programu.

Wady:

- Subskrybenci powiadamiani są w przypadkowej kolejności.