Mediator

Mediator to wzorzec, który pozwoli Ci zredukować chaos zależności w systemie. Ogranicza on bezpośrednią komunikację między obiektami, zmniejszając coupling i zmuszając obiekty do współpracy wyłącznie z mediatorem.

Jeśli chciałbyś stworzyć system do zarządzania wydarzeniami w aplikacji - to może być dobre rozwiązanie!

Jeśli szukasz narzędzia, które pozwoli Ci na sterowanie wieloma paczkami, albo ich orkiestrację - to może być to czego szukasz!

Pomyślmy o standardowym projekcie aplikacji. Bezpośrednio rozmawiające ze sobą komponenty mogą sprawić, że Twój kod będzie trudny do reużywania. Mediator sugeruje przerwanie bezpośredniej komunikacji między komponentami które mają być niezależne. Każdy komponent będzie jednak od teraz współpracować pośrednio, wywołując specjalny obiekt mediatora, który przekierowuje wywołania do odpowiednich komponentów.

To sprawia, że komponenty zależą tylko od pojedynczej klasy mediatora, zamiast sprzężenia ze sobą nawzajem.

Czy budowałeś kiedyś system, ze świadomością, że może się on rozrosnąć? Masz wtedy wiele możliwości. Możesz to zignorować, ale w przyszłości przyjdzie za to zapłacić. Możesz od razu zabrać się za budowanie mikroserwisów i skomplikowanej architektury. Problem polega jednak na tym, że na początku nawet dobrze nie znasz i nie rozumiesz domeny biznesowej. Czy jest coś pomiędzy? Tak! Można skorzystać z modularnego monolitu. Można go zrealizować na przykład za pomocą poszczególnych paczek composera. Gdy nadejdzie potrzeba - taka paczka mogłaby stać się niezależną aplikacją (mikroserwisem).

Tutaj pojawia się jednak pewien problem - żeby faktycznie było to możliwe, musisz zadbać o to by poszczególne komponenty nie były ze sobą powiązane. I właśnie tutaj, cały na biało wchodzi mediator.

Nasze wyzwanie polegać będzie na połączeniu ze sobą 3 niezależnych modułów, które muszą współpracować zależnie od siebie. Prowadzimy sklep internetowy. Po zamówieniu należy poinformować magazyn o złożonym zamówieniu, przetworzyć faktyczną płatność a także wysłać odpowiednie powiadomienie do użytkownika.

Mamy więc 3 klasy realizujące konkretne zadania, nie miejące ze sobą bezpośrednich zależności.

```
class OrdersModule {
    private Mediator $mediator;
    private array $productsId;

    public function __construct(private string $id, int ...$p
        $this->productsId = $productId;
    }

    public function setMediator(Mediator $mediator) {
        $this->mediator = $mediator;
    }

    public function placeOrder() {
        foreach ($this->productsId as $productId) {
            echo "Dodano produkt o id $productId do zamówieni
        }

        $this->mediator->notify('order:placed', ['productsId'
    }
}
```

Moduł zamówień można utworzyć podając jego ID oraz poszczególne produkty, które są zamawiane. Metoda placeOrder faktycznie odpowiada za składanie zamówienia, po czym informuje jasno mediatora o tym, że zamówienie zostało złożone.

Zajmijmy się kolejnym modułem - modułem płatności.

```
class PaymentModule {
    private Mediator $mediator;

    public function __construct(private string $currency, pri
    }

    public function setMediator(Mediator $mediator) {
        $this->mediator = $mediator;
```

```
    }

    public function addProductToPay(int $productId): void
    {
        $productPrice = match($productId) {
            1 => 100,
            2 => 200,
            3 => 300
        };

        echo "Dodano produkt o id $productId do płatności na

        $this->amount += $productPrice;
    }

    public function processPayment() {
        echo "Opłacanie zamówienia w walucie $this->currency

        $this->mediator->notify(
            'payment:processed',
            ['currency' => $this->currency, 'amount' => $this
        );
    }
}
```

Moduł płatności, posiada dwie metody. Po pierwsze przetwarzanie płatności, po drugie pobieranie cen i dodawanie ich do sumy całkowitej. Zauważ, że choć korzystamy tutaj z produktów, nie mamy pojęcia o istnieniu modułu zamówień. Wiemy tylko o identyfikatorach produktów i to na ich podstawie pobieramy ceny.

Ten moduł mógłby robić jeszcze wiele innych, związanych z ceną rzeczy, na przykład stosowanie rabatów, kodów promocyjnych, obliczanie strategii kup 2 zapłać za jedną itd. Nadal jednak nie mamy pojęcia o zamówieniu - bo nie musimy go mieć!

Po złożeniu i przetworzeniu płatności, cena i kwota zostaną przekazane mediatorowi.

Kolejnym modułem będzie moduł notyfikacji.

```
class NotificationModule {
    private Mediator $mediator;
    private string $orderId;
    private array $products;
    private float $price;
    private string $currency;

    public function __construct(private string $email) {
    }

    public function setMediator(Mediator $mediator) {
        $this->mediator = $mediator;
    }

    public function sendNotification() {
        echo "Na adres ".$this->email." wysłano wiadomość o z

        $this->mediator->notify('notification:sent', ['email'
    }

    public function getMessage(): string
    {
        $products = join("\n", array_map(fn(string $product) =

        return "Zamówienie ID: ".$this->orderId."\n".$product
    }

    public function addOrder(string $orderId, array $products
    {
        $this->orderId = $orderId;
        $this->products = $productsId;
    }

    public function setPrice(float $price, string $currency):
    {
        $this->price = $price;
        $this->currency = $currency;
```

```
    }
}
```

Moduł notyfikacji, wysyłać musi szczegółowe informacje do naszego klienta. Tak naprawdę, interesować nas będą tutaj wszystkie informacje. ID zamówienia, ID poszczególnych produktów, cena całkowita czy waluta jaką płacimy.

Mimo to, nie chcemy mieć jakiegokolwiek couplingu na moduł zamówienia i faktycznie nie mamy.

Moduł udostępnia metody `addOrder` oraz `setPrice` by móc ustawić poszczególne parametry. Ich wartości są przejrzyste, no może poza drobnym $productsId bo PHP nie pozwala narazie na typowanie tablic.

Nie musimy jednak absolutnie wiedzieć czegokolwiek o pozostałych modułach, ani o tym jak działają. Tym wszystkim zajmie się nasz mediator!

Mediator będzie implementował interfejs

```
interface Mediator
{
    public function notify(string $event, array $payload): vo
}
```

A sam jako konkretna klasa, będzie zawiadywał całym projektem.

```
class OrderMediator implements Mediator
{
    public function __construct(
        private OrdersModule       $ordersModule,
        private PaymentModule      $paymentModule,
        private NotificationModule $notificationModule
    )
    {
        $this->ordersModule->setMediator($this);
        $this->paymentModule->setMediator($this);
        $this->notificationModule->setMediator($this);
    }

    public function notify(string $event, array $payload): vo
```

```
    {
        switch ($event) {
            case 'order:placed':
                echo "Po złożeniu zamówienia przechodzę do pr
                foreach($payload['productsId'] as $productId)
                    $this->paymentModule->addProductToPay($pr
                }

                $this->notificationModule->addOrder($payload[
                break;
            case 'payment:processed':
                echo "Po zakończeniu procesu płatności przech
                $this->notificationModule->setPrice($payload[
                break;
        }
    }
}
```

Mamy więc w nim konkretne instancje naszych modułów (które mogły by również zostać przekazane przez interfejsy) jak również implementację metody notify. Otrzyma ona z poszczególnych modułów dane nas interesujące, a następnie przekaże je tam gdzie chcemy.

I tak po złożeniu zamówienia paymentModule otrzyma poszczególne produkty do dodania ceny, a moduł notyfikacje informacje o zamówieniu.

Z kolei po przetworzeniu zamówienia cena i waluta zostaną przekazane do poinformowania użytkownika.

Wzorzec mediator warto stosować, gdy zależy Ci na ograniczeniu zależności między modułami, albo na rozwiązaniu pozwalającym generować eventy. Niekiedy zmiana kas bywa też trudna, ze względy na coupling już istniejący - wtedy zastosowanie mediatora może być bardzo pomocne!

Jeśli tworzysz bardzo dużą ilość podklas danego komponentu, to również wskazuje, że może lepsze byłoby użycie mediatora.

System Eventów jest znany od dawna i to również jest rozwiązanie, zapewniające obsługę kodu w efektywny sposób. Laravel czy Symfony mają

tego typu narzędzia wbudowane, jednak jeśli chcesz stworzyć własne - nie ma z tym żadnego problemu.

Teraz wiesz jak to zrobić!

Wady i zalety:

Zalety:

- *Zasada pojedynczej odpowiedzialności*. Możesz wyekstrahować komunikację pomiędzy różnymi komponentami w jedno miejsce, czyniąc ją łatwiejszą do zrozumienia i utrzymania.
- *Zasada otwarte/zamknięte*. Można wprowadzać kolejnych mediatorów bez konieczności zmiany samych komponentów.
- Można zredukować sprzężenie pomiędzy różnymi komponentami programu.
- Można ułatwić ponowne wykorzystanie komponentów.

Wady:

- Z czasem mediator może ewoluować do postaci Boskiego Obiektu.