Command

Wzorzec projektowy Polecenie, to wzorzec, który może Ci pomóc w orkiestowaniu Twojej aplikacji. Poszczególne komendy odpowiedzialne są za poszczególne akcje, podczas gdy Invoker będzie zajmował się ich wywołaniem.

Dzięki temu żądanie zamieniamy w samodzielny obiekt. Zawiera on wszystkie informacje o tym co ma się wydarzyć, dzięki czemu możemy poszczególne elementy wywołać natychmiast, później a nawet jako kolejkę następujących po sobie działań.

Za takie działanie odpowiada klasa Invoker która wykona poszczególne nasze komendy i procedury po kolei. Oczywiście dzięki abstrakcji nie koniecznie musi je znać i wiedzieć jakie konkretnie kroki są wykonywane. Za to odpowiedzialny będzie klient, przypisujący poszczególne działania tak jak mu wygodnie. A skoro poszczególne komendy można serializować, to nic nie stoi na przeszkodzie, by je odroczyć.

Jednocześnie komendy nie powinny być odpowiedzialne za wykonywanie logiki. Jeśli jest ona prosta, to można to zaakceptować, ale jeśli jest bardziej złożona, tym powinna zająć się osobna klasa.

Nie myl jednak wzorca tego wzorca z CQS czy CQRS. Komendy co prawda nie powinny zwracać danych, natomiast celem tutaj jest możliwość sterowania określonymi akcjami, co zrobimy za chwilę w naszym zadaniu.

Chcemy stworzyć proste rozwiazanie do wysyłania notyfikacji. Wiadomości będziemy zapisywać, a następnie wyślemy odpowiednie notyfikacje. Jakie? To zależy od przypadku.

Zanim zaczniemy, stwórzmy właściwy interfejs.

```
interface Command
{
    public function execute(): void;
}
```

Jest dość prosty - nie przyjmuje niczego i łatwo go zrealizować.

Aby możliwy był zapis do pliku, stworzymy komendę która na to pozwoli:

```
class FileStorageCommand implements Command
{
    public function __construct(private string $message)
    {
    }

    public function execute(): void
    {
        echo "Store message to file: $this->message" . PHP_EO
        file_put_contents('message.txt', $this->message);
    }
}
```

Jeśli nie będziemy chcieli niczego zapisywać, a tylko zapisać dane do pamięci użyjemy innego polecenia:

```
class InMemoryStorageCommand implements Command
{
    public function __construct(private string $message)
    {
    }

    public function execute(): void
    {
        echo "Memory has message: $this->message" . PHP_EOL;
    }
}
```

Storage już mamy. Oczywiście moglibyśmy pokusić się o specjalne interfejsy dla zapisu, ale w tym momencie nie ma to znaczenia. Zauważ że obie komendy wykonują proste zadania i same z siebie stanowią małe programy.

Na podobnej zasadzie stworzymy rozwiązania do wysyłania wiadomości.

```
class MailMessageCommand implements Command
{
    public function __construct(private string $email, privat
    {
```

```
    }

    public function execute(): void
    {
        echo "Send message to email: $this->email" . PHP_EOL;
        mail($this->email, 'New message', $this->message);
    }
}
```

Mamy mail message, ale w przypadku slacka będzie to nieco bardziej skomplikowana procedura. Komendy powinny odpowiadać za wykonanie, ale już niespecjalnie za logikę biznesową.

W tym celu skorzystamy z odwracania zależności i do konstruktora przekażemy usługę, bezpośrednio odpowiedzialną za powiadomienia Slack.

```
class SlackHook
{
    public function __construct(private string $hook)
    {
    }

    public function send(string $message): void
    {
        echo "POST $message to " . $this->hook . PHP_EOL;
    }
}
```

Komenda będzie wyglądała w ten sposób.

```
class SlackMessageCommand implements Command
{
    public function __construct(private SlackHook $hook, priv
    {
    }

    public function execute(): void
    {
        echo "Send message to slack: ".$this->message . PHP_E
```

```
        $this->hook->send($this->message);
    }
}
```

Co nam to dało?

Mamy 4 komendy, które możemy łączyć w dowolny sposób. Jeśli chcemy stworzyć inne metody komunikacji, wystarczy stworzyć odpowiednią komendę. Jeśli chcemy zapisać dane w inny sposób, również stworzymy odpowiednią komendę.

Za przeprowadzenie całego procesu odpowiedzialny będzie nasz Invoker

```
class CreateMessageInvoker
{
    private Command $storeCommand;
    private Command $notifyCommand;

    public function setStoreCommand(Command $store): void
    {
        $this->storeCommand = $store;
    }

    public function setNotifyCommand(Command $notify): void
    {
        $this->notifyCommand = $notify;
    }

    public function __invoke(): void
    {
        $this->storeCommand->execute();

        if(isset($this->notifyCommand) && $this->notifyComan
            $this->notifyCommand->execute();
        }
    }
}
```

A skoro mamy nasz invoker, który przy okazji możemy wywoływać tak jak funkcję, czas go uruchomić.

W swojej aplikacji poszczególnymi komendami możesz zarządzać jak tylko chcesz. Jeśli chciałbyś wysyłać więcej powiadomień, mógłbyś robić to za pomocą tablicy powiadomień. Invoker odpowiada za przeprowadzanie kolejnych procedur, ale jakich procedur? To już jest zależne od klienta.

Kod wywołania może w związku z tym wyglądać bardzo różnie:

```
$handler = new CreateMessageInvoker();

$message = "First message";
$handler->setStoreCommand(new FileStorageCommand($message));
$handler->setNotifyCommand(new MailMessageCommand('marcin@len

$handler();
```

Zapis do pliku + wysłanie maila? Żaden problem

```
$handler = new CreateMessageInvoker();

$message = "Second message";
$handler->setStoreCommand(new InMemoryStorageCommand($message
$slackHook = new SlackHook('https://hooks.slack.com/services/
$handler->setNotifyCommand(new SlackMessageCommand($slackHook

$handler();
```

Wysłanie wiadomości za pomocą Slacka, bez zapisy do pamięci? Gotowe.

```
$handler = new CreateMessageInvoker();

$message = "Third message";
$handler->setStoreCommand(new FileStorageCommand($message));
// no notify command

$handler();
```

Alternatywnie, możemy nie powiadamiać o zapisie w żaden sposób.

Wzorzec pozwala na bardzo elastyczne działanie aplikacji. Jeśli chcesz parametryzować obiekty za pomocą działań, wzorzec ten może być dobrym

wyborem. Polecenia możesz wykonywać jako argumenty metod.

Dzięki temu możesz też układać kolejki zadań, harmonogramy i uruchamiać je zdalnie, jeśli zajdzie taka potrzeba. Dodatkowo jeśli chcesz możesz sprawić, że Twoje operacje będą odwracalne, zapisując poprzedni stan.

Zalety i wady

Zalety

- Zasada pojedynczej odpowiedzialności. Można rozprzęgnąć klasy wywołujące polecenia od klas faktycznie je wykonujących.
- Zasada otwarte/zamknięte. Można wprowadzić nowe polecenia do aplikacji bez psucia istniejącego kodu klienta.
- Pozwala zaimplementować cofnij/ponów.
- Pozwala zaimplementować opóźnione wykonywanie działań.
- Można złożyć zestaw prostszych poleceń w jedno skomplikowane.

Wady

- Kod może stać się bardziej skomplikowany gdyż wprowadzamy całą nową warstwę pomiędzy nadawcami a odbiorcami.