Strategia

Wzorzec tak popularny, że istnieje spora szansa, że znasz go od dawna. W kursie SOLIDa omówiłem wzorzec strategia na całe 3 sposoby. Nie chodzi tutaj o sposób implementacji - bo tych jest naprawdę sporo. Chodzi o zastosowanie odpowiedniej klasy dla odpowiedniej sytuacji.

Strategia pozwala uspójnić określone zestawy klas i traktować je wymiennie. Poszczególne strategie zazwyczaj niewiele o sobie wiedzą, dzięki czemu nie musi między nimi istnieć jakakolwiek zależność.

I tutaj też warto wspomnieć o DRY. Jeśli jeszcze o nim nie słyszałeś, to DRY jest zasadą, by kod nie był powtarzalny. Don’t repeat yourself. To co do zasady dobra reguła, ale doświadczeni programiści tworzą reguły po to by je łamać. W przypadku Strategii używanie DRYa może Ci wyjść na niekorzyść. Każda z klas strategii moim zdaniem powinna być niezależna. Jeśli jednak wykonują one generyczne zadanie dla systemu, oczywiście możesz posłużyć się inną klasą, realizującą określoną operację.

Strategia zawsze wykonywana jest na danym kontekście, jednak faktyczny wybór strategii istnieje zazwyczaj po stronie klienta. W większości wypadków na podstawie określonego requesta, możesz przekazać pasującą do niego klasę, choćby korzystając z mapowania.

Ja uważam jednak, że sama strategia może być śmiało wybierana przez kontekst, po przez korzystanie z listy dostępnych dla niego klas.

Strategia w najprostszej formie, może być wręcz trudna do uchwycenia - ot po prostu przekazanie instancji klasy właściwej klasie. Takie zwykłe komponowanie klas. Zobaczmy, jak mogłoby to wyglądać.

Nasze zadanie będzie polegać na stworzeniu koszyka, w którym będziemy mogli aplikować różnego typu promocje.

Standardowy klient zapłaci zwykłą cenę, jednak w naszym sklepie mamy też czasami promocję, polegającą na tym, że kupując dwa takie same produkty, płacimy za tylko jeden.

Zacznijmy od encji produktu - mamy tutaj SKU i cenę.

```
class Product
{
```

```
    public function __construct(
        private string $sku,
        private float $price
    )
    {
    }

    public function getSku(): string
    {
        return $this->sku;
    }

    public function getPrice(): float
    {
        return $this->price;
    }
}
```

Sam koszyk wyglądać będzie dość prosto, mamy tutaj wyłącznie kalkulację ceny.

```
class Cart
{
    public function __construct(
        private array $products,
        private Strategy $strategy
    )
    {
        if(count(array_filter($this->products, fn($product) =>
            throw new InvalidArgumentException('Products must
        }
    }

    public function calculatePrice(): float
    {
        return $this->strategy->getPrice($this->products);
    }
}
```

Wymaga ona jednak podania strategii tego, w jaki sposób aktualnie funkcjonuje nasz sklep. Każde strategia współdzieli interfejs `Strategy`.

```
interface Strategy
{
    public function getPrice(array $products): float;
}
```

Mając go możemy zaprogramować strategię standardową

```
class StandardCustomerStrategy implements Strategy
{
    public function getPrice(array $products): float
    {
        return array_sum(array_map(fn($product) => $product->
    }
}
```

oraz strategię promocyjną

```
class BuyTwoPayOneStrategy implements Strategy
{
    public function getPrice(array $products): float
    {
        $map = [];
        $count = [];
        $price = 0;

        foreach($products as $product) {
            $map[$product->getSku()] = $product;
            $count[$product->getSku()] = ($count[$product->ge
        }

        foreach($count as $sku => $amount) {
            $price += ceil($amount / 2) * ($map[$sku]->getPri
        }

        return $price;
```

```
    }
}
```

Najpierw mapuje i liczy ona produkty, potem kalkuluje odpowiednią cenę. Proste.

Teraz możemy użyć naszego kodu, na dwa różne sposoby.

Mamy tablicę produktów i możemy łatwo kalkulować cenę dla standardowych zakupów, oraz dla tych, gdy promocja jest włączona.

```
$products = [
    new Product('AAA', 10),
    new Product('AAA', 10),
    new Product('BBB', 30),
    new Product('AAA', 10)
];

$standardCustomerCart = new Cart($products, new StandardCusto
echo 'Standard price: ' . $standardCustomerCart->calculatePri

$standardCustomerCart = new Cart($products, new BuyTwoPayOneS
echo 'Discount price: ' . $standardCustomerCart->calculatePri
```

Zauważ jednak, że mamy tutaj po prostu odwrócenie zależności, o której mówiłem wcześniej. Musimy jednak w praktyce wybrać tą strategię od ręki.

Alternatywnie moglibyśmy umieścić logikę wyboru strategii w koszyku. Zróbmy tutaj klika zmian. Niech strategia otrzyma nową metodę - `isSatisfiedBy`.

```
interface Strategy
{
    public function getPrice(array $products): float;

    public function isSatisfiedBy(): bool;
}
```

Nasza promocja działać będzie tylko w dni robocze. W związku z tym zaimplementujemy metody w poszczególnych strategiach.

```
class StandardCustomerStrategy implements Strategy
{
    //...

    public function isSatisfiedBy(): bool
    {
        return true;
    }
}
```

Standardowa strategia zawsze będzie mogła być użyta, z kolei `BuyTwoPayOneStrategy` użyta będzie mogła być tylko w dni robocze.

```
class BuyTwoPayOneStrategy implements Strategy
{
    // ...

    public function isSatisfiedBy(): bool
    {
        return date('w') <= 5;
    }
}
```

Teraz zmieńmy nasz koszyk, by sam ustalał strategię.

```
class Cart
{
    private Strategy $strategy;

    const array STRATEGIES = [
        BuyTwoPayOneStrategy::class,
        StandardCustomerStrategy::class
    ];

    public function __construct(
        private array $products
    )
    {
```

```
        if(count(array_filter($this->products, fn($product) =
            throw new InvalidArgumentException('Products must
        }

        foreach(self::STRATEGIES as $strategy) {
            $this->strategy = new $strategy();
            if($this->strategy->isSatisfiedBy()) {
                break;
            }
        }
    }

    // ... other
}
```

Tak wygląda teraz nasz koszyk - przechodzi po kolei po strategiach i sprawdza czy można je zastosować. Jeśli tak, wybiera ją i działa tak samo jak wcześniej.

W tej sytuacji oczywiście nie musimy już wywoływać konkretnej strategii, więc nasz kod wygląda następująco:

```
$products = [
    new Product('AAA', 10),
    new Product('AAA', 10),
    new Product('BBB', 30),
    new Product('AAA', 10)
];

$standardCustomerCart = new Cart($products);
echo 'Standard price: ' . $standardCustomerCart->calculatePri
```

Skuteczne prawda? Nadal koszyk nic nie wie o zasadach, ale po prostu ma zadeklarowane strategie.

Zalety i wady

Zalety

- Możesz zamieniać algorytmy stosowane w obrębie obiektu w trakcie działania programu.
- Możesz odizolować szczegóły implementacyjne algorytmu od kodu który z niego korzysta.
- Umożliwia zamianę dziedziczenia na kompozycję.
- *Zasada otwarte/zamknięte*. Możliwe jest wprowadzanie nowych strategii bez konieczności dokonywania zmian w kontekście.

Wady

- Jeśli masz zaledwie kilka algorytmów i rzadko ulegają one zmianie, nie ma wyraźnej potrzeby nadmiernego komplikowania programu przez dodawanie nowych klas i interfejsów związanych z tym wzorcem.
- Klienci muszą być świadomi różnic pomiędzy poszczególnymi strategiami, aby mogli wybrać właściwą.
- Wiele nowoczesnych języków programowania posiada wsparcie dla typów funkcyjnych pozwalających zaimplementować różne wersje algorytmu wewnątrz zestawu anonimowych funkcji. Można następnie korzystać z tych funkcji dokładnie tak jak z obiektów strategia, ale bez konieczności rozbudowy kodu o kolejne klasy i interfejsy.