Decorator

Dziedziczenie ma swoje problemy. Zamiast niego, często lepiej wykorzystać kompozycję.

Czasem jednak i kompozycja na nic się nie zda. Co jeśli potrzebujemy by klasa wykonywała i zagnieżdżała kolejne, zależne od siebie operacje?

Może się wydawać, że w tym wypadku dziedziczenie wypadnie świetnie, tylko... Co się stanie, jeżeli kolejność będzie musiała być zmieniona? Co jeśli chcemy ją zmieniać i mieć wpływ na to co po kolei się wydarzy?

Oczywiście możemy skorzystać z kolejnych wywołań kodu, ale... Co jeśli nie chodzi nam o wynik, a o dostęp do tej docelowej klasy?

Tutaj z pomocą może nam przyjść bardzo prosty w implementacji wzorzec o nazwie dekorator.

A wiesz kto najbardziej “dekoruje” nasz świat? Politycy.

Niezależnie od tego z jakiej partii, ostatecznie i tak chodzi o to żebyśmy płacili podatki. A te bywają... skomplikowane. I na rozwiązaniu tego skomplikowania, będzie opierać się nasz dekorator.

Ale zanim skorzystamy z Dekoratora, musimy stworzyć interfejs, który zaimplementujemy do naszych pierwszych klas.

```
interface Tax
{
    public function calculate(float $price): float;

    public function getName(): string;
}
```

Prosta sprawa - kalkulator i pobieranie nazwy.

Dobrze by było, gdyby wystarczające było policzenie VAT, ale weźmy taką spółkę. Nasz programista zarabia miesięcznie 200.000zł + wszystkie podatki -

dużo, w końcu inwestuje w odpowiednie kursy żeby się rozwijać... Jeśli wiesz o czym mówię ;)

A ile wynoszą te podatki? To zależy od tego jak je policzymy i który wariant wybierzemy.

W związku z tym, nie możemy wykonać standardowego dziedziczenia na naszych klasach.

Zacznijmy od stworzenia bazowej klasy, która doda dla nas VAT.

```
 class Vat implements Tax
{
    public function calculate(float $price): float
    {
        return $price * 1.23;
    }

    public function getName(): string
    {
        return "VAT";
    }
}
```

Vat, niezależnie od wszystkiego, zawsze zapłacimy ten sam. Podatek jednak konstruuje się dodając do niego kolejne i kolejne sumy. Poza VAT będziemy mieć jeszcze CIT i ostatecznie podatek dochodowy.

W przypadku dochodowego, wszystko zależy od tego czy programista rozlicza się umową o dzieło, czy ma standardową umowę o pracę ze spółką.

No ni jak nie da się tego zrobić standardowym dziedziczeniem, bo oczekujemy przynajmniej 2 odpowiedzi.

W związku z tym stwórzmy klasę abstrakcyjną TaxDecorator

```
abstract class TaxDecorator implements Tax
{
    protected Tax $tax;

    public function __construct(Tax $tax)
    {
```

```
        $this->tax = $tax;
    }

    public function calculate(float $price): float
    {
        return $this->tax->calculate($price);
    }

    abstract public function getName(): string;
}
```

Klasa ta będzie otrzymywała kolejne dekoratory i aplikowała je jako rozwiązanie. Co więcej, każdy z dekoratorów będzie ją implementował.

Czas przejść zatem do implementacji poszczególnych podatków. Zacznijmy od CITu.

CIT płaci się od zysku, więc w tym wypadku potrzebujemy kapitału zakładowego i kosztów. Nasz dekorator zatem otrzymuje dodatkowo konstruktor.

```
class CitTax extends TaxDecorator
{
    public function __construct(Tax $tax, private float $shar
    {
        parent::__construct($tax);
    }

    public function calculate(float $price): float
    {
        $price = parent::calculate($price);
        $taxToBeCalculated = $price - $this->shareCapital - $
        if($taxToBeCalculated <= 0) return $price;

        return $price + ($taxToBeCalculated * 0.19);
    }

    public function getName(): string
```

```
    {
        return "CIT";
    }
}
```

Jeśli nie ma zysku, zwracamy obecną kwotę, jeśli jest, obniżamy ją o 19%.

No dobra, a co z dochodowym. Tutaj mamy 2 możliwości.

Pierwsza, standardowa:

```
class IncomeTax extends TaxDecorator
{
    public function calculate(float $price): float
    {
        if(parent::calculate($price) >= 120000) {
            return parent::calculate($price) * 1.32;
        }

        return parent::calculate($price) * 1.12;
    }

    public function getName(): string
    {
        return "INCOME";
    }
}
```

Jeśli przekroczyliśmy 120.000 to zapłacimy 32%, jeśli nie, jedynie 12%. No chyba że skorzystamy z umowy o dzieło, z przekazaniem praw autorskich. Taki podatek będzie kalkulowany nieco inaczej.

```
class IncomeTaxForCopyrightTransfer extends TaxDecorator
{
    public function calculate(float $price): float
    {
        $price = parent::calculate($price);
        return $price + (($price * 0.5) * 0.12);
    }
```

```
    public function getName(): string
    {
        return "INCOME";
    }
}
```

Zauważ, że wszystkie te klasy robią jedną rzecz. Poza bazowym VATem, wszystko tutaj sprowadza się do udekorowania klasy. Na start przyjmujemy klasę którą chcemy udekorować, czyli inaczej “owrapować”, a na koniec wykonujemy dekoratora, realizując jego efekty.

Jak to wygląda w przypadku implementacji?

```
function calculate(Tax $tax, int $price)
{
    echo $tax->getName() . " Tax: " . $tax->calculate($price)
}

$tax = new Vat();
$cit = new CitTax($tax, 5000, 1000);

// Price with standard income tax
$income = new IncomeTax($cit);
calculate($income, 200000);

// Price with tax for copyright transfer
$income = new IncomeTaxForCopyrightTransfer($cit);
calculate($income, 200000);
```

Na początek mamy podatek VAT pod zmienną $tax. Następnie przekazujemy tą klasę, a precyzyjniej dekorujemy tą klasę, klasą CitTax.

Nic nie stoi jednak na przeszkodzie, żeby dekoratora, ponownie udekorować.

W ten sposób stosujemy zarówno `IncomeTax` jak i `IncomeTaxForCopyrightTransfer`. Obie te klasy otrzymują dekorator cit jako wartość wejściową, dzięki czemu możemy przeprowadzić dwie operacje.

Czyli w przypadku standardowego podatku dochodowego klasa `Vat` jest dekorowana za pomocą `CitTax` by udekorować się właśnie `IncomeTax`.

W przypadku podatku z przekazaniem praw autorskich wszystko działa analogicznie, ale ostatecznym dekoratorem jest tu klasa `IncomeTaxForCopyrightTransfer`.

Czy całość nie brzmi jak dziedzicznie? Brzmi, ale nie jesteśmy nim zablokowani i to właśnie istota dekoratora.

Wady i zalety

Zalety:

- Można rozszerzać zachowanie obiektu bez tworzenia podklasy.
- Można dodawać lub usuwać obowiązki obiektu w trakcie działania programu.
- Możliwe jest łączenie wielu zachowań poprzez nałożenie wielu dekoratorów na obiekt.
- *Zasada pojedynczej odpowiedzialności*. Można podzielić klasę monolityczną, która implementuje wiele wariantów zachowań, na mniejsze klasy.

Wady:

- Zabranie jednej konkretnej nakładki ze środka stosu nakładek jest trudne.
- Trudno jest zaimplementować dekorator w taki sposób, aby jego zachowanie nie zależało od kolejności ułożenia nakładek na stosie.
- Kod wstępnie konfigurujący warstwy może wyglądać brzydko.