Kompozyt

Chcesz stworzyć strukturę drzewiastą w PHP? Najlepszym do tego rozwiązaniem będzie kompozyt. Nie myl go jednak z zasadą kompozycja nad dziedziczenie. Kompozyt, choć korzysta z wstrzykniętych klas, służy do tworzenia drzew zależności, które mogą być traktowane jednorodnie, bez względy na to czy są to pojedyncze obiekty czy nie.

Kompozyt możemy podzielić zasadniczo na trzy części.

- Komponent - to wspólny interfejs dla wszystkich obiektów w strukturze.
- Liść - reprezentuje pojedynczy obiekt. Nie ma podobieństwa do struktur złożonych.
- Gałąź - reprezentuje złożony obiekt, który składa się z innych obiektów. Zarówno liści jak i innych gałęzi.

I teraz istota kompozytu. Traktuje on jednakowo zarówno liście jak i gałęzie. Obsługa takich klas będzie wyglądała więc analogicznie. Dzięki temy możemy dodawać tyle gałęzi ile chcemy, ale samą obsługę traktować tak jak się nam podoba.

Sprawdźmy to w praktyce!

Naszym zadaniem będzie stworzenie narzędzia do planowania struktury organizacji. A organizacje, jak to organizacje, mają swoich pracowników, działy i szefów. Jednocześnie każdy z nich jest pracownikiem i powinien być w stanie wykonywać podobne operacje.

I taką strukturę, bardzo łatwo możemy umieścić w kompozycie. Jeśli nasze elementy są wzajemnie od siebie zależne, tworząc większą całość czegoś, co ma podobne właściwości, możemy wykorzystać ten właśnie wzorzec.

Zacznijmy więc od wspólnego interfejsu.

```
abstract class Employee
{
    protected ?Employee $parent;
```

```
    public function setManager(?Employee $parent): void
    {
        $this->parent = $parent;
    }

    public function getManager(): Employee
    {
        return $this->parent;
    }

    public function hire(Employee $component): void
    {
    }

    public function fire(Employee $component): void
    {
    }

    public function isManager(): bool
    {
        return false;
    }

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

Każdy pracownik, niezależnie od tego czy jest managerem czy pracownikiem szeregowym, będzie implementował klasę abstrakcyjną `Employee`. Zaprogramujmy więc szeregowego pracownika.

```
class SingleEmployee extends Employee
{
    public function __construct(private string $name)
    {
    }

    public function info(): string
```

```
    {
        return $this->name;
    }
}
```

A następnie managera, który będzie miał nieco więcej możliwości.

I tutaj, manager będzie właśnie naszym kompozytem. Za pomocą parametru `subordinates` i metody `hire` przypiszemy zależnego pracownika. Zauważ, że nie interesuje nas, czy jest to pracownik szeregowy, czy również manager.

```
class Manager extends Employee
{
    protected \SplObjectStorage $subordinates;

    public function __construct(private string $name)
    {
        $this->subordinates = new \SplObjectStorage();
    }

    public function hire(Employee $component): void
    {
        $this->subordinates->attach($component);
        $component->setManager($this);
    }

    public function fire(Employee $component): void
    {
        $this->subordinates->detach($component);
        $component->setManager(null);
    }

    public function isManager(): bool
    {
        return true;
    }

    public function info(): string
    {
```

```
        $results = [];
        foreach ($this->subordinates as $child) {
            $results[] = $child->info();
        }

        return $this->name . " manages (" . implode(", ", $re
    }
}
```

W takim wypadku Manager jest naszą kompozytową gałęzią, do której można przypisać większą liczbę pracowników. Jest jednak również pracownikiem, podobnie jak wszyscy którzy zostaną do niego przypisani.

Metoda info pomaga nam w zdefiniowaniu informacji o tym, kto podlega danemu pracownikowi.

Możemy ją wywołać korzystając z funkcji `structure`

```
function structure(Employee $component): void
{
    echo "Struktura: " . $component->info();
}
```

Wszystko gotowe, więc możemy teraz stworzyć naszą firmę.

Szefem wszystkich szefów będzie Mariusz

```
$ceo = new Manager("Mariusz");
```

Jego pierwszą decyzją będzie zatrudnienie Krzyśka - szefa technologii, oraz Kasi - szefowej sprzedaży.

```
$technology = new Manager("Krzysiek");
$ceo->hire($technology);

$sales = new Manager("Kasia");
$ceo->hire($sales);
```

Firma oczywiście nie składa się z samych managerów, więc Krzysiek zatrudnia od razu 2 developerów, a Kasia swoją koleżankę, Anię która będzie jej pomagać

```
$phpDeveloper = new SingleEmployee("Wojtek");
$technology->hire($phpDeveloper);

$reactDeveloper = new SingleEmployee("Kuba");
$technology->hire($reactDeveloper);

$juniorSales = new SingleEmployee("Ania");
$sales->hire($juniorSales);
```

Od czasu do czasu, firma będzie też potrzebować pomocy grafika. On jednak nie będzie nikomu podlegał, będzie freelancerem.

```
$design = new SingleEmployee("Janek");
```

Zauważ, że zdefiniowany designer, może w ogóle nie być częścią całej struktury. Bez przypisania, łatwo możemy korzystać z jego klasy, nie dając mu nawet żadnych zależności.

Teraz mając już poskładaną firmę, możemy wyświetlić naszą strukturę.

```
echo "Cała firma:\n";
structure($ceo);
echo "\n\n";

echo "Dział technologii:\n";
structure($technology);
echo "\n\n";

echo "Dział sprzedaży:\n";
structure($technology);
echo "\n\n";

echo "Dział designu:\n";
structure($design);
echo "\n\n";
```

Co da nam rezultat:

```
Cała firma:
Struktura: Mariusz manages (Krzysiek manages (Wojtek, Kuba), |

Dział technologii:
Struktura: Krzysiek manages (Wojtek, Kuba)

Dział sprzedaży:
Struktura: Krzysiek manages (Wojtek, Kuba)

Dział designu:
Struktura: Janek
```

Fajne, proste i przyjemne. Struktura firmy jest dość prosta, ale jednocześnie ma ogromny potencjał do określonych działań naszych klas.

Pomyśl jednak, że kompozytu możesz użyć na przykład do uzależniania od siebie elementów interfejsu. tworzenia bardziej zaawansowanych zależności, konfigurowania złożonych produktów, które w zależności od poszczególnych elementów mogą zmieniać cenę i wielu, wielu innych elementów. To chyba najlepszy sposób implementacji struktury drzewiastej w Twojej aplikacji.

Kod pozostaje czysty i nie ma couplingu między poszczególnymi klasami, za wyjątkiem klasy abstrakcyjnej.

Wady i zalety

Zalety

- Można pracować ze skomplikowanymi strukturami drzewiastymi w wygodny sposób: wykorzystaj na swoją korzyść polimorfizm i rekursję.
- Zasada otwarte/zamknięte. Możesz wprowadzać do programu obsługę nowych typów elementów bez psucia istniejącego kodu, gdyż pracuje on teraz z drzewem różnych obiektów.

Wady

- Ustalenie wspólnego interfejsu dla klas o diametralnie różnych funkcjonalnościach może okazać się trudne. W pewnych przypadkach trzeba przesadnie uogólnić interfejs komponentu, co uczyni go trudniejszym do zrozumienia.