

Metoda szablonowa

Korzystasz z dziedziczenia? W takim razie wysoce prawdopodobne jest, że korzystasz z metody szablonowej!

Metoda szablonowa to wzorzec projektowy, który pozwala wykorzystać zdefiniowany szkielet algorytmu, po to, by możliwe było jego wykorzystanie jako szkieletu.

Programując w PHP, dobrze znane powinny być Ci takie elementy jak klasy i metody abstrakcyjne.

Korzystamy tutaj ze słowa kluczowego `abstract`, która definiuje, że określona klasa może być wyłącznie dziedziczona. Obok niej znajdują się również klasy typu `final` - czyli takie, które nie mogą być rodzicami.

Zastanówmy się przez chwilę, czy to w całości nie rozwiązuje zagadnienia o którymś mowa - przecież właśnie klasa abstrakcyjna może wykonywać całą, szablonową logikę, a konkretne klasy ją dziedziczące, mogą z łatwością wykorzystać jej strukturę i oczekiwania, by implementacja metody szablonowej była prosta.

Jeśli to coś, co jest Ci znajome - to masz rację - możesz korzystać z metody szablonowej, nawet nie zdając sobie z tego sprawy.

Zanim jednak przejdziemy do wyzwania, chcę Cię przed czymś ostrzec. Nie ma nic gorszego, niż szeroka struktura i głębokie dziedziczenie, które w gruncie rzeczy nie jest potrzebne.

Klasa dziedzicząca klasę, która dziedziczy inną klasę, która dziedziczy jeszcze inną klasę, to przepis nie tylko na wysoki `coupling`, ale również na poważne problemy, związane z utrzymaniem kodu w przyszłości.

Często, zamiast korzystać z metody szablonowej i dziedziczenia, warto jest użyć kompozycję.

Istnieje nawet taka zasada - `composition over inheritance`. Czy napewno chcesz tworzyć długi sznur klas, czy może lepiej za pomocą `dependency inversion` przekazać im określone cechy w kodzie.

Ten temat poruszaliśmy już w kursie SOLIDa i odpowiada za niego literka `D` tego akronimu.

My jednak skoncentrujemy się na metodzie szablonowej. Kiedy warto ją wdrożyć? Wtedy, gdy świadomie chcemy to zrobić i potrzebujemy żeby struktura określonej klasy praktycznie zawsze wyglądała tak samo.

Naszym przypadkiem, niech będzie proste narzędzie do zapisywania treści do persystencji. Znamy przecież ogromną ilość możliwych do zapisania metod. JSON, CSV, relacyjna baza danych z wieloma silnikami, czy bazy nierelacyjne.

Chcemy mieć proste narzędzie, do którego będziemy mogli ładować nasze dane - obecnie wyłącznie je tworząc.

Zanim je jednak utworzymy, nie chcemy dopuścić do tego, że zapiszemy jakiegokolwiek puste rekordy. Co więcej, normalne treści będziemy również chcieli skompresować.

Trudność pojawia się jednak dopiero w momencie, w którym te same dane chcielibyśmy móc zapisywać do różnych nośników. Niech będą to plik JSON, CSV oraz XML.

I tutaj z łatwością możemy utworzyć klasę Storage, która będzie realizować nasz proces za pomocą metody store.

```
abstract class Storage
{
    protected array $data = [];

    abstract protected function build(): string;

    abstract protected function getFilename(): string;

    public function store(): void
    {
        $this->filter();
        $preparedData = $this->build();
        $compressed = $this->compress($preparedData);
        $this->write($compressed);
    }

    public function addToStorage(?string $payload): void
    {
        $this->data[] = $payload;
    }
}
```

```

protected function filter(): void
{
    $this->data = array_filter($this->data, fn($item) => :
}

protected function compress(string $preparedData): string
{
    return gzcompress($preparedData);
}

protected function write(string $compressed): void
{
    file_put_contents($this->getFilename(), $compressed);
}

}

```

Zauważ, że nasza klasa jest abstrakcyjna, więc wymaga klasy, która skonkretyzuje jej zadania.

Metoda odpowiedzialna za zapis - `store` będzie zapisywać dane zgodnie z wcześniejszym procesem.

Najpierw je odfiltruje, szukając pustych elementów, następnie zbuduje odpowiednią strukturę, by na koniec ją skompresować i zapisać do pliku.

Ani metoda `build`, ani metoda `getFilename` nie jest jednak przez nią oznaczona - tym zajmować się będą konkretne klasy implementujące określony typ danych.

Zacznijmy od JSONa - sprawa dość prosta, jako że PHP ma wbudowane do tego narzędzia:

```

final class JsonStorage extends Storage
{
    protected function build(): string
    {
        return json_encode($this->data);
    }

    protected function getFilename(): string

```

```

    {
        return 'data.json';
    }
}

```

Użycie typu final jest tutaj nieprzypadkowe - świadomie nie chcemy umożliwić budowania rozbudowanej struktury. Zwracamy nazwą pliku, oraz budujemy dane za pośrednictwem metody json_encode.

Kolejnym sposobem zapisu jest CSV - ten rodzaj pliku ma to do siebie, że poszczególne dane, mogą być oddzielone różnego rodzaju separatorami. Średnik, przecinek, cokolwiek - dlatego chcemy dać tutaj możliwość określenia, w jaki sposób chcemy zapisać nasze dane.

```

final class CsvStorage extends Storage
{
    public function __construct(private string $delimiter = '
    {
    }

    protected function build(): string
    {
        return implode($this->delimiter, $this->data);
    }

    protected function getFilename(): string
    {
        return 'data.csv';
    }
}

```

Co prawda zawsze będziemy mieć tylko jeden rekord, jednak zauważ, że możemy stworzyć instancję klasy CSV w bardzo prosty sposób, a jej zapis również nie jest w żadnym wypadku problematyczny.

Natomiast na koniec pozostaje nam XML. Z jakiegoś powodu, stwierdziliśmy że XMLa nie będziemy chcieli kompresować - i tutaj właśnie pojawia się pewien problem, związany z metodą szablonową. Musimy bowiem napisać nieco bezsensowny fragment kodu, związany z kompresją.

Zobaczcie

```
final class XmlStorage extends Storage
{
    protected function build(): string
    {
        $xml = new SimpleXMLElement('<root/>');
        foreach ($this->data as $item) {
            $xml->addChild('item', $item);
        }
        return $xml->asXML();
    }

    protected function getFilename(): string
    {
        return 'data.xml';
    }

    protected function compress(string $preparedData): string
    {
        return $preparedData;
    }
}
```

To właśnie dlatego, kompozycja nad dziedziczenie, może być w niektórych wypadkach lepsza - nie potrzebujemy pisać pustego kodu, albo takiego który po prostu nic nie robi.

Nie mniej jednak jak widzimy, metoda szablonowa pozwala nam w bardzo prosty sposób podkładać za kod klienta to, co faktycznie nas interesuje, w zależności od typu możemy w ten sam sposób korzystać z metod i mieć pewność, że zostaną one nie tylko poprawnie odfiltrowane, zbudowane, skompresowane i zapisane - ale także, że jeśli zajdzie potrzeba zmiany - będziemy mogli to zrobić w bardzo prosty sposób.

A jak wyglądać może faktycznie wykonywany kod?

```
function clientCode(Storage $storage): void
{
    $storage->addToStorage('some data');
```

```
$storage->addStorage(null);
$storage->addStorage('something else');
$storage->store();
}

clientCode(new JsonStorage());
clientCode(new CsvStorage(';'));
clientCode(new XmlStorage());
```

Prawda że proste i wygodne?

Wady i zalety

Zalety:

- Można pozwolić klientom nadpisać tylko niektóre partie dużego algorytmu czyniąc go odporniejszym na szkody wskutek zmian poszczególnych jego części.
- Można przenieść powtarzający się kod do klasy bazowej.

Wady:

- Dla niektórych klientów przygotowany szkielet algorytmu może stanowić ograniczenie.
- Może prowadzić do naruszenia *Zasady podstawienia Liskov* wskutek stłumienia domyślnych implementacji etapów w podklasach.
- Metody szablonowe zwykle trudniej utrzymywać w miarę jak przybywa etapów.