Most / Bridge

Już we wzorcu Adapter wstępnie powiedziałem Ci o abstrakcji. O ile adapter służy nam do tego, by dostosować niezbyt pasujące klasy do kształtu systemu. O tyle most służy do tego, by tworzyć połączenie między abstrakcją a implementacją.

Sama w sobie abstrakcja powinna obejmować logikę kontrolną wysokiego poziomu.

Uprośćmy to.

Chodzi o to, by określone działania dało się przeprowadzać za pomocą dowolnych, wybranych później, konkretnych implementacji.

Taki przykład mieliśmy we wzorcu adapter:

```
function createThenUpdate(SocialMediaPublisher $publisher, st
{
    $id = $publisher->create($content);
    $publisher->edit($id, $content . ' Update: resolved');
}
```

Nie wiedzieliśmy czym dokładnie jest publisher. W samej abstrakcji nas to nawet nie interesowało. Fakt, SocialMediaPublisher brzmi jak coś, co zawierać będzie narzędzie do publikowania w mediach społecznościowych, ale równie dobrze moglibyśmy go nazwać Publisher i wiedzieć o nim jeszcze mniej.

Jednocześnie, nawet niezbyt wprawne oko, będzie w stanie przeczytać, co tu się właściwie wydarzyło. Coś utworzyliśmy, a potem coś wyedytowaliśmy. I tak - ten kod jest głupi. Przecież od ręki moglibyśmy utworzyć właściwą treść, zamiast później ją aktualizować.

To również jest zaleta abstrakcji. Ułatwia ona zrozumienie procesu, czasami pokazując, że operacje można wykonać w bardziej elegancki sposób.

Fizycznym przykładem takiej abstrakcji może być pilot. O czym pomyślałeś? Być może o pilocie do soundbara? Być może o pilocie do zmiany koloru światła? A być może o pilocie do telewizora.

Ostatecznie, to co konkretnie będzie on obsługiwał, nie ma znaczenia. Pilot może być bardzo podobny. Zawiera jakieś przyciski, zawiera możliwość włączenia i wyłączenia. Dla przykładu taki interfejs w kodzie, mógłby wyglądać następująco:

```
interface Device
{
    public function isEnabled(): bool;

    public function enable(): void;

    public function disable(): void;

    public function getVolume(): int;

    public function setVolume(int $percent): void;

    public function getChannel(): int;

    public function setChannel(int $channel): void;
}
```

Interfejs jest bardzo prosty. Każde urządzenie może z niego korzystać.

Telewizor możemy włączyć czy wyłączyć. Zgłośnić czy zmienić kanał. Bardzo podobnie będzie z radiem, komputerem czy światłem.

Abstrakcja może jednocześnie mieć swoje rozszerzenie. Na przykład możemy mieć pilota, który będzie zdolny do sterowania za pomocą ruchu kursora. Nie każde urządzenie będzie pozwalać na tego typu obsługę. To nie szkodzi, bo tego typu interfejs może być rozszerzeniem interfejsu.

Warto przy tej okazji przypomnieć, jakie cechy mają interfejsy.

Po pierwsze faktycznie możemy używać dziedziczenia.

```
interface PointerDevice extends Device
{
    public function move(int $x, int $y): void;

    public function click(): void;
```

```
    public function getMove(): array;
}
```

Po drugie, możemy korzystać z więcej niż jednego interfejsu, co dotyczy również samego interfejsu.

W przypadku klasy wyglądało by to następująco:

```
class Television implements Device, PointerDevice
{
    // ... implementation
}
```

A w przypadku interfejsu tak:

```
interface BothFeatures extends Device, PointerDevice
{
}
```

Tutaj ważna uwaga. Ani `ComputerRemoteDevice` ani `BothFeatures` nie potrzebują definicji obu interfejsów, jeśli korzystają z tego dziedziczonego. PHPStorm z odpowiednimi ustawieniami, nawet Ci to pokaże.

Przejdźmy teraz do wzorca brigde. Stwórzmy abstrakcję do obsługi różnych urządzeń. Cechą wzorca bridge jest to, że umożliwia on niezależne zmiany zarówno implementacji, jak i warstwy abstrakcyjnej.

Skoro mamy już nasz telewizor, to poprawmy w nim interfejsy, tak by implementowany był już bezpośrednio PointerDevice. Sprawmy także by nasz telewizor zaczął faktycznie “działać”.

```
class Television implements PointerDevice
{
    private bool $isEnabled = false;
    private int $volume = 50;
    private int $channel = 1;
    private array $move = [0, 0];

    public function isEnabled(): bool
```

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

    public function enable(): void
    {
        echo "Turning on the television\n";
        $this->isEnabled = true;
    }

    public function disable(): void
    {
        echo "Turning off the television\n";
        $this->isEnabled = false;
    }

    public function getVolume(): int
    {
        return $this->volume;
    }

    public function setVolume(int $percent): void
    {
        echo "Setting volume to " . $percent . "\n";
        $this->volume = $percent;
    }

    public function getChannel(): int
    {
        return $this->channel;
    }

    public function setChannel(int $channel): void
    {
        echo "Setting channel to " . $channel . "\n";
        $this->channel = $channel;
    }
```

```
    public function move(int $x, int $y): void
    {
        echo "Moving to position: " . $x . ", " . $y . "\n";
        $this->move = [$x, $y];
    }

    public function click(): void
    {
        echo "Clicking on position: " . implode(', ', $this->m
    }

    public function getMove(): array
    {
        return $this->move;
    }
}
```

Klasa telewizora gotowa. Nasz telewizor obsługuje też kursor.

Na podobnej zasadzie stwórzmy teraz klasę radia.

```
class Radio implements Device
{
    private bool $isEnabled = false;
    private int $volume = 50;
    private int $channel = 1;

    public function isEnabled(): bool
    {
        return $this->isEnabled;
    }

    public function enable(): void
    {
        echo "Turning on the radio\n";
        $this->isEnabled = true;
    }

    public function disable(): void
```

```
    {
        echo "Turning off the radio\n";
        $this->isEnabled = false;
    }

    public function getVolume(): int
    {
        return $this->volume;
    }

    public function setVolume(int $percent): void
    {
        echo "Setting radio volume to " . $percent . "\n";
        $this->volume = $percent;
    }

    public function getChannel(): int
    {
        return $this->channel;
    }

    public function setChannel(int $channel): void
    {
        echo match($channel) {
            1 => "Setting radio channel to RMF\n",
            2 => "Setting radio channel to Z\n",
            3 => "Setting radio channel to Eska\n",
            default => "Setting radio to ".$channel."MHz\n",
        };
        $this->channel = $channel;
    }
}
```

Nie obsługuje ona jednak wskaźnika. Taki byłby bezsensowny w przypadku radia.

Pierwszą część, czyli konkretną implementację, mamy gotową. Czas na część abstrakcyjną - pilota. I choć on również mógłby implementować Device - w tym konkretnym wypadku nie będziemy tego robić.

Istotą wzorca most, jest to, że może być niezależna. Dlatego na potrzeby naszej lekcji pilota pozostawimy niezależnego. Nie będzie on mógł poinformować nas jak głośno gra urządzenie, czy też który kanał jest właśnie ustawiony.

```
class Remote
{
    public function __construct(private Device $device)
    {
    }

    public function turnOn(): void
    {
        $this->device->enable();
    }

    public function turnOff(): void
    {
        $this->device->disable();
    }

    public function setChannel(int $channel): void
    {
        $this->device->setChannel($channel);
    }

    public function setVolume(int $percent): void
    {
        $this->device->setVolume($percent);
    }
}
```

I tak przygotowany most możemy już wykorzystać.

```
function runAndChangeChanel(Device $device)
{
    $remote = new Remote($device);
    $remote->turnOn();
```

```
}

runAndChangeChanel(new Television());
runAndChangeChanel(new Radio());
```

Ponownie znamy tutaj interfejs `Device` nasza funkcja tworzy nowego pilota, który następnie przyjmie różnego rodzaju urządzenia, a następnie je włączy, zmieni kanał na 1 i zgłośni go do 50%.

W ten sposób zobaczymy następujące komunikaty:

```
Turning on the television
Setting channel to 1
Setting volume to 50

Turning on the radio
Setting radio channel to RMF
Setting radio volume to 50
```

Dzięki oddzielenia abstrakcji od implementacji, Builder pozwala nam również zwiększyć liczbę dostępnych wariantów. Kto powiedział, że możemy mieć tylko jednego pilota?

Chcemy stworzyć mysz komputerową, która będzie miała odpowiednie przyciski? Zero problemu!

```
class Mouse
{
    public function __construct(private Device $device)
    {
    }

    public function run(): void
    {
        $this->device->enable();
        $this->device->setVolume(100);
    }

    public function move(int $x, int $y): void
    {
```

```
        if(!($this->device instanceof PointerDevice)) return;

        $this->device->move($x, $y);
    }

    public function click(): void
    {
        if(!($this->device instanceof PointerDevice)) return;

        $this->device->click();
    }
}
```

Nasza mysz może teraz włączyć dowolne urządzenie, ale sterować wyłącznie tymi, które wykorzystują kursor.

Możemy zatem uruchomić nieco inny kod:

```
function runAndMove(Device $device)
{
    $remote = new Mouse($device);
    $remote->run();
    $remote->click();
    $remote->move(100, 200);
}

runAndMove(new Television());
echo PHP_EOL;
runAndMove(new Radio());
```

Który ostatecznie wywoła

```
Turning on the television
Setting volume to 100
Clicking on position: 0, 0
Moving to position: 100, 200
```

```
Turning on the radio
Setting radio volume to 100
```

Zauważ, że kod funkcji `runAndMove` i `runAndChangeChanel`, kod poszczególnych urządzeń i poszczególnych abstrakcji możemy rozwijać w pełni niezależnie.

Most umożliwia nam rozdzielenie i przeorganizowanie monolitycznych klas, posiadających wiele wariantów tej samej funkcjonalności, w kod który może być bardzo zwinny. Potrzebujesz różnych wariantów i opcji, łączonych ze sobą klas? Jak widzisz Bridge pozwala to robić naprawdę łatwo a do tego pozostawić odpowiednie funkcjonalności niezależne sobie. Nie znające swoich zależności.

Możemy dzięki temu rozszerzać nasze klasy na wielu, niezależnych płaszczyznach. Jednocześnie spełniając wymóg wyboru implementacji, w trakcie działania programu.

Wady i zalety

Zalety

- Możesz tworzyć niezależne od platformy klasy i aplikacje.
- Kod klienta działa na wyższym poziomie abstrakcji. Nie musi mieć do czynienia ze szczegółami platformy.
- *Zasada otwarte/zamknięte*. Możesz wprowadzać nowe abstrakcje i implementacje niezależnie od siebie.
- *Zasada pojedynczej odpowiedzialności*. W abstrakcji możesz skupić się na wysokopoziomowej logice, zaś w implementacji na szczegółach platformy.

Wady

- Kod może stać się bardziej skomplikowany gdy zastosuje się ten wzorzec w przypadku wysoce zwartej klasy.