Adapter

Adapter to świetny wzorzec projektowy, w szczególności jeśli przyjdzie Ci pracować z kodem legacy. Bo wiesz, świat dzieli się na ludzi, którzy absolutnie nie chcą pracować z kodem legacy i takich, którzy potrafią programować. A dług - cóż - trzeba umieć obsługiwać.

Adapter to wzorzec, który służy do zmiany interfejsu jednej klasy na interfejs innego, który będzie z nią kompatybilny.

Jeśli więc do naszej dyspozycji mamy istniejący kod, który po prostu nie działa tak, jakbyśmy chcieli - to właśnie adapter pomoże nam stworzyć rozwiązanie, bez konieczności modyfikowania kodu, którego ruszyć nie powinniśmy.

Jeśli więc chcesz tworzyć swoje rozwiązanie, w oparciu o kod legacy, którego nigdy, ale to nigdy nie należy zmieniać... Adapter jest dla Ciebie!

Adapter to wzorzec strukturalny, dlatego zatrzymajmy się na chwilę i porozmawiajmy o abstrakcji. Odnosi się do procesu ukrywania szczegółów implementacyjnych i prezentowania jedynie istotnych informacji użytkownikowi lub innym elementom systemu.

Jeśli więc chcesz tworzyć naprawdę czysty kod - chowaj konkretną implementację za interfejsami, albo klasami abstrakcyjnymi.

Przejdźmy do naszego zadania. Tak się jakoś kiepsko złożyło, że znalazłeś się w świecie kodu legacy, który wymaga naprawy. Jest jednak pewien warunek - nie wolno Ci modyfikować starej klasy, bo wiele innych elementów z niej korzysta.

Nasza “odziedziczona” klasa wygląda w ten sposób:

```
class MySpacePublisher
{
    private $token;

    public function __construct($token)
    {
    }
```

```
    public function post($content)
    {
        echo "Sending POST method with content " . $content .
        return uniqid();
    }

    public function delete($postId)
    {
        echo "Sending DELETE method to " . $postId;
    }
}
```

Nasz kod nie jest jakiś bardzo tragiczny, choć brakuje w nim typowania, a sama platforma nie ma możliwości aktualizacji - bo jej nie zaimplementowano.

Nasze zadanie jest dość proste. Chcemy stworzyć system, który umożliwi publikację, aktualizację i możliwość publikowania postów na wielu platformach social media.

Ich ilość będzie rosła, a my chcemy utrzymać kod tak, by w łatwy sposób można było go rozwijać.

Jeśli istniała by możliwość modyfikacji kodu - pierwsze co należało by zrobić, to dodać interfejs lub klasę abstrakcyjną, która zapewniła by nam spójny sposób komunikacji. Zasady to jednak zasady - nie da się - to korzystamy z adaptera. Zanim jednak to się stanie stwórzmy interfejs, z którego chcielibyśmy docelowo korzystać.

```
interface SocialMediaPublisher
{
    public function create(string $content): string;
    public function edit(string $id, string $content): void;
    public function delete(string $id): void;
}
```

Prosty CRUD, bez odczytywania. Mając taką klasę, możemy sobie wyobrazić nieco lepszą implementację publikacji postów na instagrama.

```
class InstagramPublisher implements SocialMediaPublisher
{
    public function create(string $content): string
    {
        echo "Creating Instagram post with content " . $conte
        return uniqid();
    }

    public function edit(string $id, string $content): void
    {
        echo "Editing Instagram post with id " . $id . " and
    }

    public function delete(string $id): void
    {
        echo "Deleting Instagram post with id " . $id . "\n";
    }
}
```

To nasz hipotetyczny kod.

Teraz czas na prostą funkcję do implementacji:

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

Proste, choć łamie SRP i jest bez sensu - ale dla naszego przykładu jest ok!

No i dobra - wszystko bardzo fajnie - tylko jak użyć naszego kodu legacy. Po pierwsze nie ma on metody edit, po drugie - nie chcemy pisać jakiejkolwiek ifologii, zmierzającej do `if(MySpacePublisher)` then... W ten sposób stworzylibyśmy ogrmny coupling na to, że skorzystaliśmy z takiego, a nie innego rozwiązania. Co jeśli pojawi się inne, które będziemy preferować? Co jeśli będziemy w stanie napisać odpowiednią klasę osobiście, albo... po prostu w całości z niej zrezygnować.

Ify zasadniczo nie są dobre - bo trzeba je testować dla obu przypadków. W tym wypadku możemy sobie poradzić bez nich, korzystając właśnie z adaptera.

Zaimplementujmy więc nową klasę, która będzie adapterem dla `MySpacePublisher` jednocześnie implementującą `SocialMediaPublisher`.

```
class MySpaceAdapter implements SocialMediaPublisher
{
    public function __construct(private MySpacePublisher $myS
    {
    }

    public function create(string $content): string
    {
        return $this->mySpacePublisher->post($content);
    }

    public function edit(string $id, string $content): void
    {
        $this->mySpacePublisher->delete($id);
        $this->mySpacePublisher->post($content);
    }

    public function delete(string $id): void
    {
        $this->mySpacePublisher->delete($id);
    }
}
```

Proste, nie? Zauważ, że klasa `MySpacePublisher` nie ulega zmianie, ale jest przekazywana jako konstruktor. Czemu tak, a nie na przykład przez dziedziczenie?

Po pierwsze dziedziczenie zasadniczo nie jest zbyt dobre, w szczególności gdy klasa dziedziczy klasę, która jest dziedziczona przez inną klasę. W związku z tym lepiej korzystać z kompozycji. Czyli słynne “composition over inheritance”.

Po drugie klasa `MySpacePublisher` zawiera już metodę delete, ale nie jest ona otypowana. W kodzie legacy tych metod może być bardzo dużo i o wiele

bezpieczniej jest zrobić to w ten sposób.

Teraz czas na implementację.

```
createThenUpdate(new InstagramPublisher(), 'Hello world');
echo "\n";
createThenUpdate(new MySpaceAdapter(new MySpacePublisher('tok
```

Zauważ, że klasa `MySpacePublisher` została wywołana w ten sam sposób, co zwykle. A kod `createThenUpdate` jest teraz spójny z całą resztą. W zależności od wybranej przez użytkownika metody możemy teraz używać interfejsów i aktualizować kod tak długo, jak będzie korzystał z interfejsu `SocialMediaPublisher`.

Możemy nawet skorzystać z matcha który na podstawie przekazanego publishera wyśle zadany tekst:

```
function run(string $publisher, string $content) {
    $publisherInstance = match($publisher) {
        'instagram' => new InstagramPublisher(),
        'myspace' => new MySpaceAdapter(new MySpacePublisher(
        default => throw new InvalidArgumentException('Invali
    };

    createThenUpdate($publisherInstance, $content);
}

run('instagram', 'Hello world');
run('myspace', 'Hello world');
```

Przydatne, nie?

Kod się nie zmienia, ale struktura na jakiej nam zależy, została uspójniona i to bez jednego if'a!

Adapter również świetnie nadaje się, gdy chcesz wykorzystać ponownie wiele istniejących podklas którym brakuje jakiejś wspólnej funkcjonalności. Adapter może składać je do spójnej całości, jakiej oczekujesz.

Wady i zalety:

Zalety:

- *Zasada pojedynczej odpowiedzialności*. Można oddzielić interfejs lub kod konwertujący dane od głównej logiki biznesowej programu.
- *Zasada otwarte/zamknięte*. Można wprowadzać do programu nowe typy adapterów bez psucia istniejącego kodu klienckiego, o ile będzie on korzystał z adapterów poprzez interfejs kliencki.

Wady:

- Ogólna złożoność kodu zwiększa się, ponieważ trzeba wprowadzić zestaw nowych interfejsów i klas. Czasem łatwiej zmienić klasę udostępniającą jakąś potrzebną usługę, aby pasowała do reszty kodu.