Factory

Wzorzec Factory może sprawić, że Twój kod będzie zdecydowanie bardziej elastyczny. Wyobraź sobie sytuację, w której podpinamy konkretną implementację do określonego problemu i bam - wszystko działa!

Jak i gdzie można to wykorzystać? Posłuchajcie.

Chcemy stworzyć aplikację, która będzie w stanie pobierać dane o grach komputerowych na różne platformy, a następnie je zapisywać.

Problem polega na tym, że platform udostępniających te dane jest wiele. Co więcej, nieco starsze gry, na platformy takie jak np. Amiga mamy nie w formie API, ale w formie plików excela.

Do tego wszystkiego dochodzi jeszcze jedno, poważne utrudnienie - zapis powinien odbywać się nie tylko do naszego API, ale również do innych miejsc. Nie jesteśmy w stanie zdefiniować gdzie dokładnie, ale już teraz wiemy że będą to pliki CSV.

Formatów może być naprawdę dużo, wiele z nich będzie zarówno możliwych do zapisania jak i do odczytania.

Na ten moment potrzebujemy jednak tych trzech zadań:

- Pobierania gier na xbox i zapisu do csv
- Pobierania gier na xbox i wysłania ich do naszego API
- Pobierania gier z naszego API i zapisy ich do csv

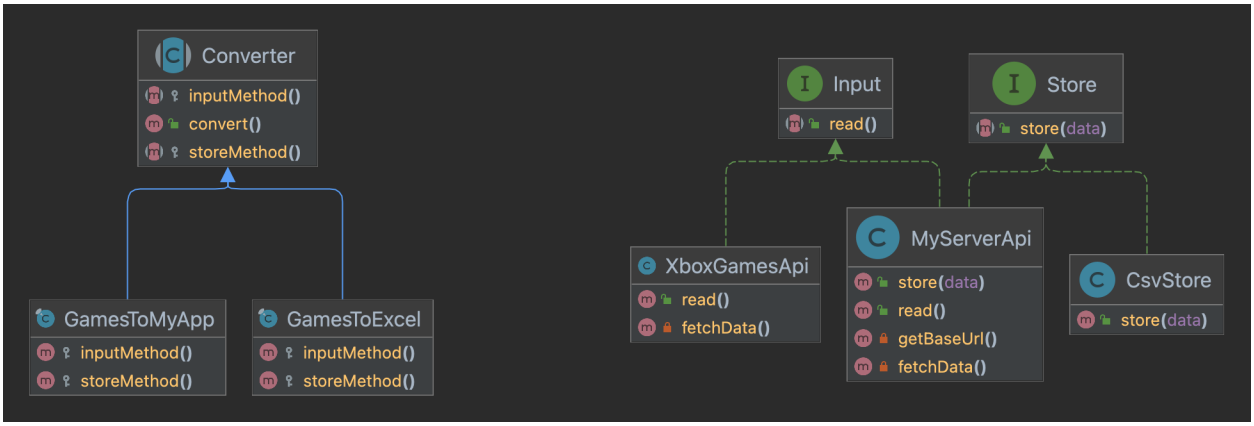
Brzmi jak potencjalnie dużo pracy?

I tutaj wchodzi wzorzec Factory - cały na biało. Udostępnia on interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów. Wzorzec Factory jest bardzo użyteczny w sytuacjach, gdy mamy do czynienia z wieloma rodzajami obiektów, a klient nie musi znać ich dokładnych implementacji.

Gdyby tak mieć konkretną klasę która zajmuje się daną warstwą, a także konsumenta który połączy wybrane ze sobą klasy.

W ten sposób pisząc łatwy kod, będziemy w stanie używać takich implementacji jakie nas interesują.

Zerknij jak wygląda struktura naszych klas:



Rozpocznijmy od implementacji abstrakcyjnej klasy Converter, dzięki czemu każda klasa będzie implementowała niezbędne do działania metody.

Innym sposobem jest sprawienie, aby bazowa metoda wytwórcza zwracała jakiś domyślny typ Input lub Store, ale wówczas byłby to pusty kod.

```
abstract class Converter
{
    abstract protected function inputMethod(): Input;
    abstract protected function storeMethod(): Store;

    public function convert(): void
    {
        $data = $this->inputMethod()->read();
        $this->storeMethod()->store($data);
    }
}
```

Klasa twórcza, implementująca w naszym wypadku Converter zwróci nam więc nowe obiekty, które użyjemy do przeprowadzenia naszego procesu. Weź jednak pod uwagę, że w realnym świecie klasa kreacyjna zawiera już jakąś ważną logikę biznesową, powiązaną z tą konwersją.

Skoro mamy stworzoną klasę Converter, czas stworzyć klasy implementujące Input i Store.

Zacznijmy od klasy `XboxGamesApi` będzie ona zajmowała się jedynie pobieraniem danych - będzie więc implementować wyłącznie interfejs Input.

```
class XboxGamesApi implements Input
{
    public function read(): array
    {
        return array_map(
            fn(array $array) => [
                $array["id"],
                $array["name"],
                $array["genre"][0] ?? '',
                $array["publishers"][0] ?? '',
            ],
            json_decode($this->fetchData(), true)
        );
    }

    private function fetchData(): string
    {
        return @file_get_contents('https://api.sampleapis.com/xl
    }
}
```

Pobieramy interesujące nas dane za pośrednictwem `file_get_contents` - to, że nie jest to zbyt optymalne rozwiązanie, nie ma teraz znaczenia. Znaczenie ma to, że zwracamy tablicę z danymi które nas interesują.

Kolejnym zadaniem będzie zaprogramowanie narzędzia, które pozwoli nam zapisywać pliki do CSV. W tym wypadku implementujemy Store.

```
class CsvStore implements Store
{
    public function __construct(private string $source)
    {
    }

    public function store(array $data): void
    {
        $fp = fopen($this->source, 'w');

        foreach ($data as $fields) {
            fputcsv($fp, $fields);
        }

        fclose($fp);
    }
}
```

Store, wraz z metodą store, przybiera tablicę danych, a następnie zapisuje ją w formacie CSV pod nazwą pliku, który podajemy przy konstruktorze.

I na sam koniec - konkretna implementacja naszego API, które może wyglądać w ten sposób:

```
class MyServerApi implements Input, Store
{
    public function read(): array
    {
        return json_decode($this->fetchData(), true) ?? [];
    }

    private function fetchData(): string
    {
```

```
        return @file_get_contents($this->getBaseUrl() . '/games
    }

    private function getBaseUrl(): string
    {
        return 'https://localhost/api';
    }

    public function store(array $data): void
    {
        $ch = curl_init($this->getBaseUrl() . '/create');

        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        curl_setopt($ch, CURLOPT_POST, true);
        curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data)

        curl_exec($ch);

        curl_close($ch);
    }
}
```

Mamy tutaj implementację dwóch interfejsów, bo to API pozwala zarówno na odczytywanie jak i zapisywanie danych.

Do tej pory wszystko wygląda dość prosto. Konkretne działanie i konkretna odpowiedź.

Dopiero teraz zaczyna się ogień, czyli łączenie ze sobą poszczególnych implementacji.

Przypomnijmy sobie te trzy zadania, które mamy zrobić:

- Pobierania gier na xbox i zapisu do csv
- Pobierania gier na xbox i wysłania ich do naszego API
- Pobierania gier z naszego API i zapisy ich do csv

Czy wiesz już jak to zaimplementować?

Tak! To bardzo proste.

```
final class GamesToExcel extends Converter
{
    protected function inputMethod(): Input
    {
        return new XboxGamesApi();
    }

    protected function storeMethod(): Store
    {
        return new CsvStore('games.csv');
    }
}
```

Input - xbox → output csv.

Teraz czas na wysłanie gier z serwera xboxa na nasz serwer.

```
final class GamesToMyApp extends Converter
{
    protected function inputMethod(): Input
    {
        return new XboxGamesApi();
    }

    protected function storeMethod(): Store
    {
        return new MyServerApi();
    }
}
```

I jak się słusznie domyślasz, stworzenie rozwiązania, zapewniającego nam pobranie gier z naszego serwera i zapis do csv będzie równie prosty... Tak prosty,

że możemy do tego nawet wykorzystać klasę anonimową!

Stwórzmy funkcję convert, która będzie wyglądała mniej więcej tak:

```
function convert(Converter $converter)
{
    $converter->convert();
}
```

Użycie naszych obecnych metod, w tym klasy anonimowej będzie wyglądać w ten sposób:

```
// This will store games in excel
convert(new GamesToExcel());

// This will store games in my app
convert(new GamesToMyApp());

// This will store data from my server in csv file
convert(new class extends Converter {
    protected function inputMethod(): Input
    {
        return new MyServerApi();
    }

    protected function storeMethod(): Store
    {
        return new CsvStore('my_games.csv');
    }
});
```

Gdzie i kiedy można zastosować ten wzorzec?

- Stosuj Metodę Wytwórczą gdy nie wiesz z góry jakie typy obiektów pojawią się w twoim programie i jakie będą między nimi zależności.
    - Tak jak w naszym przykładzie - korzystamy obecnie z CSV i 2 źródeł API. Ile źródeł danych jeszcze otrzymamy? Na jakie sposoby jeszcze będziemy się komunikować? Tego nie wiemy, ale Factory sprawia, że nie musimy się tym martwić.
    - Metoda Wytwórcza oddziela kod konstruujący produkty od kodu który faktycznie z tych produktów korzysta. Dlatego też łatwiej jest rozszerzać kod konstruujący produkty bez konieczności ingerencji w resztę kodu.
- Factory to też świetne narzędzie do rozbudowywania Twojego kodu!
    - Jeśli chcesz stworzyć możliwość realizacji pluginów do Twojej aplikacji, factory może być najlepszym do tego rozwiązaniem!
    - Poszczególne klasy można łatwo rozszerzyć i zapewnić im inne implementacje, jeśli tylko tego potrzebujesz.
- Korzystaj z Metody wytwórczej gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów, zamiast odbudowywać je raz za razem.
    - Twoje aplikacje mogą być od teraz budowane, wykorzystując małe klasy zajmujące się konkretnymi, atomowymi zadaniami które za pomocą faktorii będziesz w stanie połączyć.
    - W naszym przykładzie ilość kroków niezbędnych do wykonania to zaledwie dwa. Pobranie danych i ich zapis. Jeśli proces miałby być bardziej złożony i customizowalny - metoda wytwórcza, będzie tutaj naprawdę świetnym rozwiązaniem!

Wady i zalety rozwiązania

Zalety:

- Unikasz ścisłego sprzęgnięcia pomiędzy twórcą a konkretnymi produktami.
- *Zasada pojedynczej odpowiedzialności*. Możesz przenieść kod kreacyjny produktów w jedno miejsce programu, ułatwiając tym samym utrzymanie kodu.

- *Zasada otwarte/zamknięte*. Możesz wprowadzić do programu nowe typy produktów bez psucia istniejącego kodu klienckiego.

Wady:

- Kod może się skomplikować, ponieważ aby zaimplementować wzorzec, musisz utworzyć liczne podklasy. W najlepszej sytuacji wprowadzisz ów wzorzec projektowy do już istniejącej hierarchii klas kreacyjnych.