Abstract Factory

Czy zastanawiałeś się kiedyś, jak implementuje się naprawdę duże systemy w których jest tyle możliwych różnic? Jak stworzyć system który będzie na tyle elastyczny, że z łatwością pozwoli na ustawienie odpowiednich funkcjonalności, w zależności od wybranego typu konta... Nie tworząc przy tym karuzeli ifów, tak trudnej do utrzymywania?

Rozwiązaniem tych problemów może być wzorzec fabryki abstrakcyjnej.

Załóżmy, że chcemy rozważyć przypadek, w którym nasz użytkownik, w zależności od wybranego przez siebie pakietu, może posiadać określone funkcjonalności, ale również odpowiednie pakiety kosztują odpowiednio inną kwotę.

Albo alternatywnie, że mając określone dane - takie jak tytuł i treść, chcemy w prosty sposób wysłać wiadomość na każdy dostępny kanał, bez konieczności “pisania” maila czy ręcznego wywoływania wysyłki SMS.

W obu tych przypadkach to właśnie fabryka abstrakcyjna będzie właściwym rozwiązaniem.

Jeśli jesteś w stanie sprowadzić problem do macierzy, gdzie w jednej kolumnie znajdą się określone cechy, a w drugiej inne - to prawdopodobnie tym wzorcem projektowym, będziesz w stanie rozwiązać sprawę szybko, łatwo i przyjemnie. Oczywiście nie musimy się ograniczać do osi x i y, wymiarów możemy zrobić tyle ile chcemy. Fakt jest jednak taki - fabryka abstrakcyjna będzie wymagała od nas nieco większej ilości klas niż zazwyczaj, bo każdy element będzie musiał być opisany osobno.

Mimo to wzorzec ten jest nie tylko bardzo prosty w użyciu, ale również bardzo efektywny, bo kod można śmiało re-używać.

Fabryka abstrakcyjna definiuje interfejs służący tworzeniu poszczególnych produktów, ale pozostawia faktyczne tworzenie produktów konkretnym klasom fabrycznym. Każdy typ fabryki odpowiada jednemu z wariantów produktu.

Zaprogramujmy to!

Zakładamy, że nasza aplikacja będzie miała następujące funkcjonalności:

- możliwość wyszukiwania - która w zależności od konta, będzie się różniła wynikami (konta basic nie mają możliwości filtrowania ofert)
- możliwość zabookowania - która będzie rezerwowała pokój w standardzie zależnym of typu konta
- możliwość opłacenia - gdzie cena będzie różniła się w zależności od typu konta

Zacznijmy od zadeklarowania interfejsu, odpowiadającego fabryce abstrakcyjnej - czyli w naszym przypadku `FeatureFactory`

```
interface FeatureFactory
{
    public function search(string $query, array $filters = []):

    public function book(): BookFeature;

    public function pay(): PaymentFeature;
}
```

Oczywiście, żeby FeatureFactory miał sens, potrzebujemy poszczególnych “produktów” naszej fabryki. Z resztą z samego typowania wynika, że coś powinniśmy tutaj zadeklarować:

```
interface SearchFeature
{
    public function search(): array;
}

interface BookFeature
{
    public function booking(
        int $id,
        PaymentFeature $payment
```

```
    ): void;
}

interface PaymentFeature
{
    public function pay(): void;

    public function getAmount(): float;
}
```

To, co jest naprawdę mocne w Abstract factory, to fakt, że te poszczególne interfejsy, składające się na jedną fabrykę, będą zbudowane w oparciu o osobne klasy.

Idealnie pasuje to do sytuacji, w której chcesz zarządzać poszczególnymi funkcjonalnościami użytkownika, przesyłać innego rodzaju treść w ten sam sposób czy robić jakąkolwiek kompozycję jaką sobie zamarzysz.

No dobrze, skoro mamy już interfejsy, czas stworzyć nieco konkretnej implementacji. Zacznijmy od konta typu Basic.

```
class BasicAccount implements FeatureFactory
{
    public function search(string $query, array $filters = []):
    {
        return new BasicSearch($query, $filters);
    }

    public function book(): BookFeature
    {
        return new BasicBooking();
    }

    public function pay(): PaymentFeature
```

```
    {
        return new Payment(50);
    }
}
```

Wyszukiwanie, bookowanie czy płatności - ta konkretna faktoria otrzymuje takie klasy jakie są z nią powiązane. Nie muszą to wcale być klasy stworzone stricte dla tej konkretnej faktorii. W naszym wypadku mamy na przykład klasę Payment, która nie będzie się różnić niczym w przypadku MediumAccount, poza ważnym parametrem, jakim jest cena.

```
class MediumAccount implements FeatureFactory
{
    public function search(string $query, array $filters = []):
    {
        return new Search($query, $filters);
    }

    public function book(): BookFeature
    {
        return new StandardBooking();
    }

    public function pay(): PaymentFeature
    {
        return new Payment(100);
    }
}
```

O, a tak wygląda medium account. Widać, że zmieniła się klasa wyszukiwania oraz bookingu.

I teraz wchodzą całe na biało, poszczególne funkcjonalności, jakie będziemy implementować. Oczywiście zgodnie z interfejsami.

Zacznijmy od wyszukiwarki.

```
class BasicSearch implements SearchFeature
{
    public function __construct(private string $query, private a
    {
    }

    public function search(): array
    {
        echo "Basic search done on query ".$this->query."\n";
        return ['result' => $this->query];
    }
}
```

A w przypadku bardziej zaawansowanego wyszukiwania

```
class Search implements SearchFeature
{
    public function __construct(private string $query, private a
    {
    }

    public function search(): array
    {
        echo "Search done on query " . $this->query .
            " and filters:" . join("; ", $this->filters) . "\n",

        if(count($this->filters) > 0)
        {
            return ['result' => 'search with filters'];
        }

        return ['result' => $this->query];
    }
}
```

Oczywiście nie jest to kod faktycznie zaimplementowany, do tego dla celów późniejszej prezentacji dodane zostało wyświetlanie szczegółów, ale myślę że z samą implementacją, niemiałbyś problemów.

Bardzo analogicznie budujemy klasę `BasicBooking` oraz `StandardBooking`

```
class BasicBooking implements BookFeature
{
    public function booking(
        int $id,
        PaymentFeature $payment
    ): void
    {
        // allow basic booking
        echo "Very cheap room on id ".$id." has been paid with a
        $payment->pay();
    }
}

class StandardBooking implements BookFeature
{
    public function booking(
        int $id,
        PaymentFeature $payment
    ): void
    {
        // allow any booking
        echo "Nice room with view on id ".$id." has been paid wi
        $payment->pay();
    }
}
```

Z kolei metodę Payment, która obecnie jest wykorzystywana przez nasze oba rodzaje kont definiujemy jako jedną klasę. Myślę jednak, że jeśli tylko byś chciał -

już widzisz jak można by było zrobić całkowicie osobne rozwiązanie, do tego problemu.

```
class Payment implements PaymentFeature
{
    public function __construct(private float $amount)
    {
    }

    public function pay(): void
    {
        // pay the amount
        echo "You paid {$this->amount}\n";
    }

    public function getAmount(): float
    {
        return $this->amount;
    }
}
```

Wygląda na to że wszystko gotowe - teraz tylko pozostaje stworzyć rozwiązanie, które umożliwi nam wykorzystywać nasze funkcje, w zależności od konta. Zrobimy to jedną klasą:

```
class BookingApp
{
    public function __construct(private FeatureFactory $factory)
    {
    }

    public function search(string $query, array $filters = []):
    {
        return $this->factory->search($query, $filters)->search(
```

```
    }

    public function book(int $id): void
    {
        $payment = $this->factory->pay();

        $this->factory->book()->booking(
            $id,
            $payment
        );
    }
}
```

Metoda search, gdy zostanie używa, wywoła taką funkcję, jaka została podana w konstruktorze `BookingApp` podobnie z book, jednak w tym wypadku mamy nieco bardziej złożoną implementację, bo metoda booking otrzymuje klasę Payment z której w przypadku zabookowania pobierze odpowiednie środki.

Jak z tego korzystać? Bardzo prosto.

```
// use basic account
$app = new BookingApp(new BasicAccount());
$app->search("Portugal");
$app->book(1);

// use medium account
$app = new BookingApp(new MediumAccount());
$app->search("Portugal", ['filter1', 'filter2']);
$app->book(1);
```

Kod klienta wywołuje metody kreacyjne obiektu fabrycznego zamiast tworzyć produkty bezpośrednio — wywołując konstruktor (za pomocą operatora `new`). Skoro dana fabryka odpowiada jednemu z wariantów produktu, to wszystkie jej produkty będą ze sobą kompatybilne.

Kod klienta współpracuje z fabrykami i produktami wyłącznie poprzez ich abstrakcyjne interfejsy. Dzięki temu jeden klient jest kompatybilny z wieloma różnymi produktami. Wystarczy stworzyć nową konkretną klasę fabryczną i przekazać ją kodowi klienta.

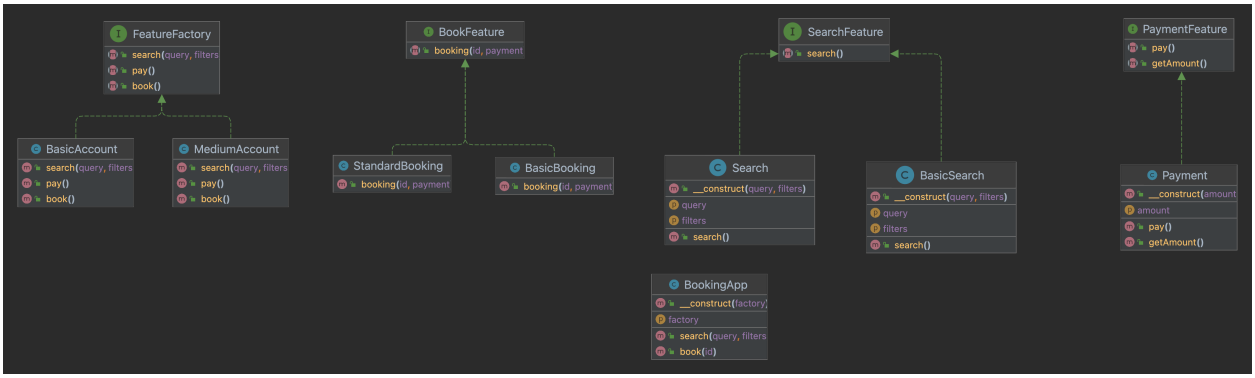
Podsumujmy sobie, co tutaj się właściwie wydarzyło.

Po pierwsze odkryliśmy, że nasza aplikacja, składająca się z funkcjonalności search, book oraz pay - jest macierzą składającą się z osi funkcjonalność / rodzaj konta.

Każda funkcjonalność musi w takiej sytuacji mieć zadeklarowany swój interfejs, co zrobiliśmy deklarując `SearchFeature`, `BookFeature` oraz `PaymentFeature`.

Następnie stworzyliśmy konkretne produkty - czyli klasy `BasicAccount` oraz `MediumAccount` implementujące interfejs `FeatureFactory`. Weź jednak pod uwagę, że oba te produkty w swoich implementacjach korzystają z interfejsów cech. Nie dlatego, by łatwo można było je wymieniać, gdyby zaszła taka potrzeba, ale również dlatego, żeby nie dochodziło do zbyt dużego couplingu.

Myślę, że dla wzorca fabryka abstrakcyjna znajdziesz bardzo wiele mądrych zastosowań i tego Ci życzę!



Gdzie i kiedy można zastosować ten wzorzec?

- **Stosuj Fabrykę abstrakcyjną, gdy twój kod ma działać na produktach z różnych rodzin, ale jednocześnie nie chcesz, aby ściśle zależał od konkretnych klas produktów. Mogą one bowiem być nieznane na wcześniejszym etapie tworzenia programu, albo chcesz umożliwić przyszłą rozszerzalność aplikacji.**
    - Fabryka abstrakcyjna dostarcza ci interfejs służący tworzeniu obiektów z różnych klas danej rodziny produktów. O ile twój kod będzie kreował obiekty za pośrednictwem tego interfejsu — nie musisz się martwić stworzeniem produktu w niezgodnym z innymi wariancie.
- Przemyśl ewentualną implementację wzorca Fabryki abstrakcyjnej, gdy masz do czynienia z klasą posiadającą zestaw Metod wytwórczych które zbytnio przyćmiewają główną odpowiedzialność tej klasy.
    - W prawidłowo zaprojektowanym programie każda klasa jest odpowiedzialna za jedną rzecz. Gdy zaś klasa ma do czynienia z wieloma typami produktów, warto być może zebrać jej metody wytwórcze i umieścić je w osobnej klasie fabrycznej, albo nawet w pełni zaimplementować Fabrykę abstrakcyjną z ich pomocą.

Wady i zalety rozwiązania

Zalety:

- Zyskujesz pewność, że produkty, jakie otrzymujesz stosując fabrykę, są ze sobą kompatybilne.
- Zapobiegasz ścisłemu sprzęgnięciu konkretnych produktów z kodem klienckim.
- *Zasada pojedynczej odpowiedzialności*. Możesz zebrać kod kreacyjny produktów w jednym miejscu w programie, ułatwiając tym samym późniejsze utrzymanie kodu.
- *Zasada otwarte/zamknięte*. Możesz wprowadzać wsparcie dla nowych wariantów produktów bez psucia istniejącego kodu klienckiego.

Wady:

- Kod może stać się bardziej skomplikowany, niż powinien. Wynika to z konieczności wprowadzenia wielu nowych interfejsów i klas w toku wdrażania tego wzorca projektowego.