Builder

Jeżeli programujesz w PHP to z pewnością korzystałeś z tego wzorca. Builder jest przecież wykorzystywany w niemalże każdej bibliotece do komunikacji z bazą danych. Bez niego nie byłoby ani Eloquenta, ani Doctrina ani innych ORMów.

Nie jest to jednak w żadnym stopniu wzorzec, który służyłby tylko do tworzenia zapytań SQL. Dzisiaj opowiem Ci o tym jak wykorzystać ten wzorzec do zupełnie innego celu - a będzie nim konfigurator produktów.

Nasz klient zajmuje się sprzedażą samochodów. Sprzedaje każdą dostępną na rynku markę i to z pełnym wyposażeniem. Teraz chciałby wprowadzić możliwość stworzenia porównywarki, która po podaniu rodzaju silnika, przebiegu i listy dodatkowego wyposażenia, zaprezentuje wszystkie auta podając ich ceny.

Inicjalizacja wszystkich tych klas w normalnych warunkach byłaby bardzo pracochłonna. Każde auto posiadało by konstruktor który może mieć inną kolejność parametrów, a zarządzanie tym, nawet przez konstruktor byłoby bardzo kłopotliwe w długim terminie. Jedna funkcja mogła by zostać z auta usunięta inna dodana - mielibyśmy spore problemy. Można by tworzyć osobną klasę dla osobnego rocznika, ale wyobraź sobie ile kodu zawierała by nasza aplikacja.

Co ważne - nie przechowujemy aut z poszczególnymi parametrami w jednej bazie. Dane mogą być rozproszone po różnych źródłach więc odpada rozwiązanie tego problemu zunifikowanym SQLem.

Czy ten problem dla się rozwiązać tak, by nie pisać wielkiego trudnego do utrzymania kodu?

Oczywiście i pomoże nam w tym wzorzec builder.

Ten wzorzec dzieli konstrukcję obiektu na poszczególne etapy. Nie musisz wywoływać ich wszystkich - więc możemy stworzyć obiekt, korzystając tylko z tych kroków, które są niezbędne a potem dodać do niego dodatkowe elementy.

Co za tym idzie, możemy tworzyć wielu budowniczych, korzystając z tych samych etapów konstrukcji. Konfigurując więc samochód, który ma mieć automatyczną skrzynię biegów, podgrzewaną kierownicę i kamerę cofania - możemy podmieniać budowniczego i wstawiać w niego różne marki. W ten sposób otrzymamy stworzony obiekt np. reprezentujący markę Jeep, Forda czy BMW.

Na podobnej zasadzie, jak się z pewnością domyślasz, mogą działać ORMy, które komunikować będą się z różnymi rodzajami baz danych w dokładnie ten sam, zunifikowany sposób.

Ostatecznym celem buildera jest zbudowanie czegoś. Tak jak w przypadku ORMa będzie to string składający się z bezpiecznego zapytania SQL, tak w naszym wypadku będzie to obiekt auta, który ostatecznie mógłby budować się sam, za pomocą parametrów, ale builder pozwoli nam również to zunifikować.

Zacznijmy od intefrejsu buildera

```
interface Builder
{
    public function getProduct(): Car;

    public function reset(): void;

        public function setEngine(EngineType $type): void;

    public function setMileage(int $mileage): void;

    public function addAccessories(Accessory $accessory): void;
}
```

Będzie on zawierał metody pozwalające na zbudowanie klasy i pobranie jej za pomocą getProduct. Będzie również, na nasze potrzeby, zawierał możliwość resetu, tak by można było auto skonfigurować ponownie.

Sam builder wyglądać będzie tak:

```
class CarBuilder implements Builder
{
    private Car $product;
    private Car $initialProduct;

    public function __construct(Car $product)
    {
        $this->initialProduct = clone $product;
```

```
        $this->reset();
    }

    public function reset(): void
    {
        $this->product = $this->initialProduct;
    }

    public function setEngine(EngineType $type): void
    {
        $this->product->setEngine($type);
    }

    public function setMileage(int $mileage): void
    {
        $this->product->setMileage($mileage);
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->product->addAccessories($accessory);
    }

    public function getProduct(): Car
    {
        return $this->product;
    }
}
```

Zauważ, że podczas tworzenia obiektu w konstruktorze przekazywane jest auto które następnie jest zapisywane jako `initialProduct`. To nie jest cecha samego buildera, ale w naszych wymaganiach zależy nam na tym, by możliwe było tworzenie auta od początku. Dlatego właśnie klonujemy klasę do parametru `initialProduct` a następnie resetujemy całość, przypisując tym samym produkt jako `initalProduct`.

Kolejnym zadaniem będzie zdefiniowanie aut. Będziemy mieć kolejno marki Jeep, Ford i BMW. Kod będzie wyglądał w ten sposób:

```
class Jeep implements Car
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }

    public function setMileage(int $mileage): void
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
    {
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "Jeep with {$this->engineType->name} engine, {$t
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.8;
                break;
```

```
            case $this->mileage > 200000:
                $uses = 0.7;
                break;
            default:
                $uses = 1;
        }

        return (match($this->engineType) {
            EngineType::AUTOMATIC => 30000,
            EngineType::MANUAL => 25000
        } + count($this->accessories) * 1000) * $uses;
    }
}

class Ford implements Car
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }

    public function setMileage(int $mileage): void
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
```

```
    {
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "Ford with {$this->engineType->name} engine, {$t
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.7;
                break;
            case $this->mileage > 200000:
                $uses = 0.6;
                break;
            default:
                $uses = 1;
        }

        return (match($this->engineType) {
                    EngineType::AUTOMATIC => 20000,
                    EngineType::MANUAL => 15000
                } + count($this->accessories) * 1200) * $uses;
    }
}

class Bmw implements Car
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }
```

```
    public function setMileage(int $mileage): void
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
    {
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "BMW with {$this->engineType->name} engine, {$thi
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.8;
                break;
            case $this->mileage > 200000:
                $uses = 0.7;
                break;
            default:
                $uses = 1;
        }

        return (match($this->engineType) {
                    EngineType::AUTOMATIC => 50000,
                    EngineType::MANUAL => 45000
                } + count($this->accessories) * 2000) * $uses;
    }
}
```

Zauważ, że same w sobie, obecne tutaj klasy, mogły by być uznane za buildera - bo faktycznie tworzą osobną klasę. I to prawda - to po części jest builder, ale w naszym wypadku zależ nam na tym, by wyjściową wartością była klasa, z którą później można by było pracować. Nie chcemy zatem umożliwiać jej resetu, a settery które w niej występują służą do enkapsulacji danych. Innymi słowy nie chcemy żeby te parametry można było zmieniać i żeby były publiczne.

Weź również pod uwagę fakt, że builder może wykonywać nieco inną logikę niż sam obiekt auta. Na przykład moglibyśmy przekazując informacje o silniku, przekazać informacje o przebiegu. Tutaj nie jest to błędem, a bardziej sposobem rozwiązania.

Chcę, żebyś zwrócił na to szczególną uwagę, bo gdybyśmy klas faktycznie nie potrzebowali to już same obiekty Car można by uznać za buildery.

Kod klas Car jest w zasadzie bardzo do siebie podobny, poza informacją o ofercie oraz pobraniem faktycznej ceny. Każda z tych implementacji otrzymuje interfejs Car. Jeśli chcesz, możesz również wykorzystać tutaj klasę abstrakcyjną, która będzie posiadała nie tylko sam kontrakt, ale również powtarzający się kod.

Nie jest to jednak w tym momencie jeszcze istotne.

Interfejs Car mogłby wyglądać następująco:

```
interface Car extends CarAttributes
{
    public function getOffer(): string;
    public function getPrice(): float;

        public function setEngine(EngineType $type): void;

    public function setMileage(int $mileage): void;

    public function addAccessories(Accessory $accessory): void;
}
```

Zauważmy jednak, że Car powtarza metody z buildera. Możemy je zatem wrzucić do osobnego interfejsu i odchudzić nieco interfejsy `Builder` oraz `Car`

```
interface CarAttributes
{
    public function setEngine(EngineType $type): void;

    public function setMileage(int $mileage): void;

    public function addAccessories(Accessory $accessory): void;
}

interface Car extends CarAttributes
{
    public function getOffer(): string;
    public function getPrice(): float;
}

interface Builder extends CarAttributes
{
    public function getProduct(): Car;

    public function reset(): void;
}
```

Oczywiście uważne oko nie przegapi, że mamy tutaj jeszcze EngineType oraz Accessory. Oba te elementy będą enumami, które definiuję następująco:

```
enum EngineType {
    case AUTOMATIC;
    case MANUAL;
}

enum Accessory {
    case GPS;
    case TRAILER;
```

```
    case RADIO;
}
```

Ok, builder zbudowany. Klasy które będą budowane również. Dobrze by było zatem użyć naszego kodu.

I tutaj mamy dwie możliwości. Zacznijmy od łatwiejszej.

```
function createCarOffer(Builder $builder)
{
    echo "Standard basic product:\n";
    $builder->setEngine(EngineType::MANUAL);
    $builder->setMileage(50000);
    echo $builder->getProduct()->getOffer();
    echo "\nPrice: ".$builder->getProduct()->getPrice()."$\n";

    echo "\n\nCustom product:\n";
    $builder->reset();
    $builder->setEngine(EngineType::AUTOMATIC);
    $builder->setMileage(0);
    echo $builder->getProduct()->getOffer();
    echo "\nPrice: ".$builder->getProduct()->getPrice()."$\n";

    echo "\n\nStandard full featured product:\n";
    $builder->reset();
    $builder->setEngine(EngineType::AUTOMATIC);
    $builder->setMileage(0);
    $builder->addAccessories(Accessory::GPS);
    $builder->addAccessories(Accessory::RADIO);
    $builder->addAccessories(Accessory::TRAILER);
    echo $builder->getProduct()->getOffer();
    echo "\nPrice: ".$builder->getProduct()->getPrice()."$\n";
}
```

Powyższa funkcja zbuduje na początek auto z manualną skrzynią i przebiegiem 50.000.

Następnie customową wersję produktu, gdzie skrzynia jest manualna a przegieg wynosi 0.

A na samym końcu produkt na pełnym wypasie. Automatyczna skrzynia, przegieg 0 i wszystkie możliwe akcesoria.

Żeby wykonać tę funkcję potrzebujemy naszego buildera. Dla każdego auta, zadeklarujemy osobnego budowaniczego.

```
createCarOffer(new CarBuilder(new Jeep()));
createCarOffer(new CarBuilder(new Ford()));
createCarOffer(new CarBuilder(new Bmw()));
```

Prawda że proste?

W ten sposób dostajemy informacje o wszystkich markach, które nas interesują, razem z ofertą i ceną. Pod `$builder->getProduct()` znajdziemy konkretny produkt, gotowego i skonfigurowanego wcześniej auta. Naszą docelową klasę.

Jeśli sobie byśmy tego życzyli, to możemy również rozwiązać ten kod, bez korzystania z wrappera, jakim jest CarBuilder. W ten sposób każda klasa marki, będzie jednocześnie builderem.

Przykład rozwiązania takiego kodu wyglądałby tak:

```
enum EngineType {
    case AUTOMATIC;
    case MANUAL;
}

enum Accessory {
    case GPS;
    case TRAILER;
    case RADIO;
}
```

```
interface Builder
{
    public function getOffer(): string;
    public function getPrice(): float;

    public function setEngine(EngineType $type): void;

    public function setMileage(int $mileage): void;

    public function addAccessories(Accessory $accessory): void;
}

class Jeep implements Builder
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }

    public function setMileage(int $mileage): void
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
    {
```

```
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "Jeep with {$this->engineType->name} engine, {$th
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.8;
                break;
            case $this->mileage > 200000:
                $uses = 0.7;
                break;
            default:
                $uses = 1;
        }

        return (match($this->engineType) {
            EngineType::AUTOMATIC => 30000,
            EngineType::MANUAL => 25000
        } + count($this->accessories) * 1000) * $uses;
    }
}

class Ford implements Builder
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }

    public function setMileage(int $mileage): void
```

```
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
    {
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "Ford with {$this->engineType->name} engine, {$t
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.7;
                break;
            case $this->mileage > 200000:
                $uses = 0.6;
                break;
            default:
                $uses = 1;
        }

        return (match($this->engineType) {
                    EngineType::AUTOMATIC => 20000,
                    EngineType::MANUAL => 15000
                } + count($this->accessories) * 1200) * $uses;
    }
}

class Bmw implements Builder
```

```
{
    private EngineType $engineType;
    private int $mileage;
    private array $accessories = [];

    public function setEngine(EngineType $type): void
    {
        $this->engineType = $type;
    }

    public function setMileage(int $mileage): void
    {
        $this->mileage = $mileage;
    }

    public function addAccessories(Accessory $accessory): void
    {
        $this->accessories[] = $accessory;
    }

    public function getOffer(): string
    {
        $accessories = join(', ', array_map(fn($accessory) => $a
        return "BMW with {$this->engineType->name} engine, {$thi
    }

    public function getPrice(): float
    {
        switch($this->mileage) {
            case $this->mileage > 100000:
                $uses = 0.8;
                break;
            case $this->mileage > 200000:
                $uses = 0.7;
                break;
            default:
```

```
                $uses = 1;
        }

        return (match($this->engineType) {
                    EngineType::AUTOMATIC => 50000,
                    EngineType::MANUAL => 45000
                } + count($this->accessories) * 2000) * $uses;
    }
}

function createCarOffer(Builder $builder)
{
    $builderInit = clone $builder;

    echo "Standard basic product:\n";
    $builder->setEngine(EngineType::MANUAL);
    $builder->setMileage(50000);
    echo $builder->getOffer();
    echo "\nPrice: ".$builder->getPrice()."$\n";

    echo "\n\nCustom product:\n";
    $builder = clone $builderInit;
    $builder->setEngine(EngineType::AUTOMATIC);
    $builder->setMileage(0);
    echo $builder->getOffer();
    echo "\nPrice: ".$builder->getPrice()."$\n";

    echo "\n\nStandard full featured product:\n";
    $builder = clone $builderInit;
    $builder->setEngine(EngineType::AUTOMATIC);
    $builder->setMileage(0);
    $builder->addAccessories(Accessory::GPS);
    $builder->addAccessories(Accessory::RADIO);
    $builder->addAccessories(Accessory::TRAILER);
    echo $builder->getOffer();
    echo "\nPrice: ".$builder->getPrice()."$\n";
```

```
}

createCarOffer(new Jeep());
createCarOffer(new Ford());
createCarOffer(new Bmw());
```

Jakie rozwiązanie lepiej pasuje do Twojego problemu - to oczywiście zależy od Ciebie. Ja w tym wypadku preferuję Builder niezależny od klas które są jego produktem, bo potem chciałbym na nich popracować.

Ciekawym sposobem może też być użycie klasy kierownika. Taka klasa mogła by być odpowiedzialna za budowanie wstępnie gotowych pakietów. W przypadku aut mamy przecież zawsze “gotowe” pakiety konfiguracyjne, więc nic nie stoi na przeszkodzie żebyśmy też napisali takie rozwiązanie.

```
class Director
{
    public function __construct(private Builder $builder)
    {
    }

    public function buildMinimalVersion(): void
    {
        $this->builder->setEngine(EngineType::MANUAL);
        $this->builder->setMileage(50000);
    }

    public function buildAutomatic(): void
    {
        $this->builder->setEngine(EngineType::AUTOMATIC);
        $this->builder->setMileage(0);
    }

    public function buildFullPackage(): void
    {
```

```
        $this->builder->setEngine(EngineType::AUTOMATIC);
        $this->builder->setMileage(0);
        $this->builder->addAccessories(Accessory::GPS);
        $this->builder->addAccessories(Accessory::RADIO);
        $this->builder->addAccessories(Accessory::TRAILER);
    }

    public function getOffer(): string
    {
        return $this->builder->getProduct()->getOffer().
            "\nPrice: ".$this->builder->getProduct()->getPrice(
    }
}
```

Klasa Director przyjmuje Buildera jako swój konstruktor, a następnie wykonuje na nim operacje, które są niezbędne.

Możemy więc napisać wywołanie naszego kodu w nieco czytelniejszy sposób. Teraz wszystkie nasze pakiety będą budowane po kolei w sposób bardziej czytelny.

```
function createCarOffer(Builder $builder)
{
    $director = new Director($builder);

    echo "Standard basic product:\n";
    $director->buildMinimalVersion();
    echo $director->getOffer();

    echo "\n\nCustom product:\n";
    $builder->reset();
    $director->buildAutomatic();
    echo $director->getOffer();

    echo "\n\nYou can still use standard builder:\n";
    $builder->reset();
```

```
    $director->buildFullPackage();
    echo $director->getOffer();
}

createCarOffer(new CarBuilder(new Jeep()));
createCarOffer(new CarBuilder(new Ford()));
createCarOffer(new CarBuilder(new Bmw()));
```

Zauważ, że dalej możemy korzystać z buildera, choćby do resetu parametrów.

Podsumujmy, co tu się właściwie wydarzyło

Na początku zdefiniowaliśmy interfejs naszego budowniczego. Zawierał on wszystkie niezbędne kroki do tego by utworzyć nasz produkt. W tym wypadku instancję konkretnej marki auta.

W przykładzie mamy użycie jednego, konkretnego budowniczego - który tworzy auto, niezależnie od marki. Tak naprawdę jednak, każde auto jest jednocześnie budowniczym i mogło by zostać potraktowane jako builder. My jednak świadomie, chcemy mieć możliwość wpływania na to jak określone produkty w przyszłości mogą być budowane i owrapowaliśmy to wszystko generycznym budowniczym.

Nasz budowniczy zwróci nam na koniec instancję klasy, zawierającą przekazane mu paramtery do budowy.

Jeśli chcemy, działanie konkretnego buildera możemy zlecić klasie Director, która będzie odpowiedzialna za budowanie gotowych “prefabrykowanych” obiektów.

Gdy kierownik otrzyma buildera, na jego podstawie zbuduje obiekt na jakim nam zależy.

Kiedy warto stosować ten wzorzec?

- Stosuj wzorzec Budowniczy, gdy potrzebujesz możliwości tworzenia różnych reprezentacji jakiegoś produktu (na przykład, domy z kamienia i domy z drewna)

  Wzorzec Budowniczy można użyć gdy konstruowanie różnorakich reprezentacji produktu obejmuje podobne etapy, które różnią się jedynie szczegółami.

Bazowy interfejs budowniczego definiuje wszelkie możliwe etapy konstrukcji, a konkretni budowniczy implementują te kroki by móc tworzyć poszczególne reprezentacje obiektów. Natomiast klasa kierownik pilnuje właściwego porządku konstruowania.

- Stosuj ten wzorzec do konstruowania drzew Kompozytowych lub innych złożonych obiektów.
  Wzorzec budowniczego umożliwia konstrukcję w etapach. Niektóre z nich możemy odroczyć bez szkody dla finalnego produktu. Możemy nawet wywoływać etapy rekursywnie, co przydaje się przy budowie drzewa obiektów.
  Budowniczy uniemożliwia dostęp do nieskończonego produktu przez okres jego konstrukcji. Zapobiega to pozyskiwaniu niekompletnych wyników przez kod kliencki.
  I tak właśnie w naszym przypadku - jeśli auto nie mogło by nie mieć przebiegu - builder może zadbać o to by taki obiekt (jeszcze nie gotowy) nie był udostępniony dalej.

Zalety i wady tego rozwiązania

Zalety:

- Możesz konstruować obiekty etapami, odkładać niektóre etapy, lub wykonywać je rekursywnie.
- Możesz wykorzystać ponownie ten sam kod konstrukcyjny budując kolejne reprezentacje produktów.
- *Zasada pojedynczej odpowiedzialności*. Można odizolować skomplikowany kod konstrukcyjny od logiki biznesowej produktu.

Wady:

- Kod staje się bardziej skomplikowany, gdyż wdrożenie tego wzorca wiąże się z dodaniem wielu nowych klas.