Design patterns

Spis treści

Czym są Design patterns?

Wzorce projektowe (**Design patterns**) są powtarzalnymi rozwiązaniami często występujących problemów, które napotkasz podczas projektowania systemów informatycznych. Nie są to fragmenty kodu, z którymi można zrobić kopiuj-wklej, raczej są to schematy/wzory/rozwiązania konkretnych powtarzalnych problemów. Schematy te można zastosować w różnych sytuacjach, natomiast to my musimy ten kod napisać, wzorując się na sprawdzonych rozwiązaniach.

Z racji, że są to rozwiązania przemyślane przez innych i sprawdzone wcześniej, to stosując takie wzorce zmniejszamy ryzyko wprowadzenia błędów, gdy sami będziemy coś implementować. Oczywiście stosując takie wzorce nadal możemy popełnić błąd w ich implementacji.

Wzorce, o których będziemy rozmawiać, są trochę jak łacina, znane wśród społeczności. Jeżeli będziemy je stosować, wiemy, że znajdziemy zrozumienie wśród innych developerów. Stosowanie znanych i popularnych wzorów rozwiązywań problemów zwiększa również czytelność kodu. Bo przecież stosujemy wzór, który jest powszechnie znany.

Wymienione poniżej wzorce odnoszą się do programowania obiektowego lub programowania zorientowanego obiektowo. Definiują one w jaki sposób możemy powiązać ze sobą klasy/obiekty aby osiągnąć jakiś efekt w sprawdzony już wcześniej sposób. I to właśnie ten konkretny sposób powiązania ze sobą klas/obiektów nazywany jest wzorcem, np. **Visitor**, **Singleton** itp.

Materiał ten jest o tyle istotny, że jeżeli zaczniesz pracować w praktyce i usłyszysz zdanie: "Słuchaj wrzuciłem tam strategię i załatwione", to będziesz w stanie zrozumieć o co chodzi. Sprowadza się to do tego, że ktoś rozwiązał problem stosując wzorzec strategii w kodzie. Jeżeli nie wiemy jak ten wzorzec wygląda to możemy nawet oglądać ten kod i nie będziemy mieli zielonego pojęcia, że został tam zastosowany jakiś wzorzec. Przykład ten pokazuje jednocześnie, że stosując wzorce projektowe możemy komunikować się z innymi developerami stosując nazwy określające rodzaje zachowań naszego programu.

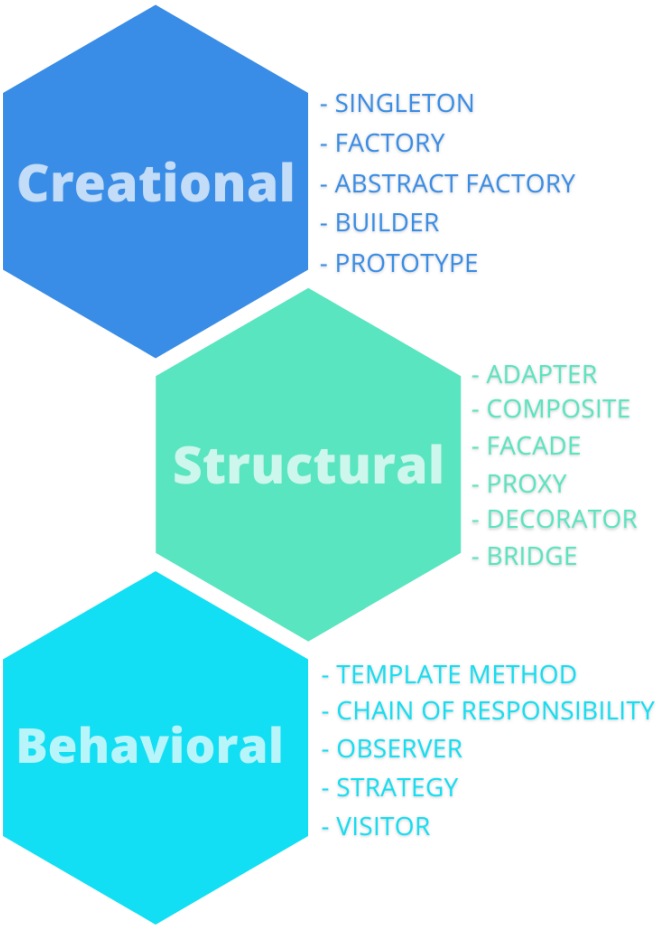
Różnica między Design Principles a Design Patterns

Ludzie często zadają sobie pytanie czym jedno różni się od drugiego (**Design Principles** od **Design Patterns**).

Design Patterns w uproszczonym języku określają nam w jaki sposób można powiązać ze sobą klasy aby osiągnąć rozwiązanie jakiegoś problemu. To ten sposób powiązania ze sobą klas nazywany jest wzorcem projektowym. Czyli jeżeli usłyszysz stwierdzenie **Visitor**, będzie to oznaczało konkretną interakcję pomiędzy klasami.

Design Principles są natomiast zasadami, których należy przestrzegać podczas pisania oprogramowania, które pomagają nam np. utrzymać czysty kod.

W kolejnych materiałach przedstawimy rodzaje **Design Patterns** w podziale na grupy oraz omówimy każdy z wymienionych. Należy oczywiście pamiętać, że wymienione przez nas wzorce projektowe nie są wszystkimi jakie istnieją. Omawiamy tylko wycinek wzorców, które są jednocześnie często spotykane w praktyce. Prawdopodobnie i tak będziesz wracać do tego tematu wielokrotnie ucząc się kolejnych, nowych wzorców lub przypominając sobie już wcześniej poznane 😀.



Obraz 1. Design patterns

Co to GoF

Jeżeli zaczniesz interesować się wzorcami projektowymi spotkasz się ze stwierdzeniem **GoF** (*Gang of Four*). Stwierdzenie to odnosi się do 4 autorów popularnej książki o tematyce wzorców projektowych. Ta książka to *"Design Patterns: Elements of Reusable Object-Oriented Software"*, a autorzy to *Erich Gamma, Richard Helm, Ralph Johnson* oraz *John Vlissides*. Często stwierdzenia **GoF** używa się w odniesieniu do książki, a nie jej autorów.

Creational Design Patterns

Ta grupa wzorców projektowych skupia się wokół tworzenia obiektów. Przykładowe wzorce to (wyjaśniamy je po kolei poniżej):

- Singleton
- Factory

- Abstract Factory
- Builder
- Prototype

Przejdźmy teraz do omówienia każdego z nich.

Singleton

Problem jaki rozwiązuje

Czasami może wystąpić taka sytuacja żeby w aplikacji istniała tylko jedna instancja danego obiektu. Nawet jeżeli będziemy próbowali stworzyć ten obiekt ponownie, zostanie nam zwrócona istniejąca już instancja.

Przykład z życia

Jakie znamy przykłady obiektów, które mogą wystąpić tylko raz? Za każdym razem gdy będziemy szukać takiego obiektu, a on istnieje to ma być zwrócony właśnie on. A jeżeli nie istnieje to należy go utworzyć? Papież albo Prezydent 😁. Papież może być tylko jeden, a jak go nie ma to należy go wybrać.

Rozwiązanie

Klasa, którą zdefiniujemy jako **Singleton** ma być odpowiedzialna za utworzenie siebie samej oraz wymuszenie istnienia tylko jednej instancji. Klasa taka ma być dostępna globalnie aby z każdego miejsca móc uzyskać instancję takiego obiektu. Teoretycznie jest to jeden z prostszych wzorców projektowych, natomiast istnieje wiele sposobów na jego implementację. Nawet jeżeli będziemy stosowali różne podejścia, należy pamiętać o kwestiach takich jak:

- konstruktor takiej klasy powinien być prywatny, aby tylko ona sama mogła decydować kiedy tworzyć obiekt,
- musimy gdzieś przetrzymywać instancję takiego obiektu, możemy to zrobić w zmiennej `private static` w tej samej klasie,
- musimy zapewnić statyczną metodę publiczną, która będzie zwracała nam instancję tej klasy,
- możemy jednocześnie taką klasę zadeklarować jako `final` żeby nie dało się z niej dziedziczyć.

`Singleton` można inaczej rozumieć jak globalną zmienną, która ma wystąpić tylko i wyłącznie raz w naszym programie. W praktyce jednak warto zwrócić uwagę, aby nie wpychać go na siłę gdy nie jest nam potrzebny.

Przykład w kodzie

Wyróżniamy kilka rodzajów Singletonów, w zależności od tego w jaki sposób je napiszemy. Skupimy się tutaj na 2: takim, który jest inicjowany w momencie wczytania klasy w Javie (możemy nazwać to **eager init**, czyli chętną inicjalizacją) oraz takim, który jest tworzony dopiero gdy tego potrzebujemy (możemy nazwać to **lazy init**, czyli leniwa inicjalizacja). Gdy przejdziemy do omówienia zagadnień

programowania wielowątkowego, powiemy też sobie o Singletonie **thread-safe**, czyli takim, który jest bezpieczny w środowisku wielowątkowym, ale to nie teraz. Pokazana implementacja jest natomiast bezpieczna o ile używamy tylko jednego wątku.

```
public final class StaticBlockSingleton {

    private static final StaticBlockSingleton instance;

    static {
        try {
            instance = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Exception occurred when creating instance");
        }
    }

    private StaticBlockSingleton() {}

    public static StaticBlockSingleton getInstance() {
        return instance;
    }
}
```

Analizując przedstawiony kod możemy zauważyć, że tworzymy instancję tego obiektu w statycznym bloku inicjalizacyjnym, zatem gdy zawołamy metodę `getInstance()`, zawsze zostanie zwrócona ta sama instancja obiektu `StaticBlockSingleton`. Nie jest to też najlepsza praktyka, bo może się okazać, że niepotrzebnie zaśmiecamy pamięć, jeżeli wcale takiego obiektu nie użyjemy.

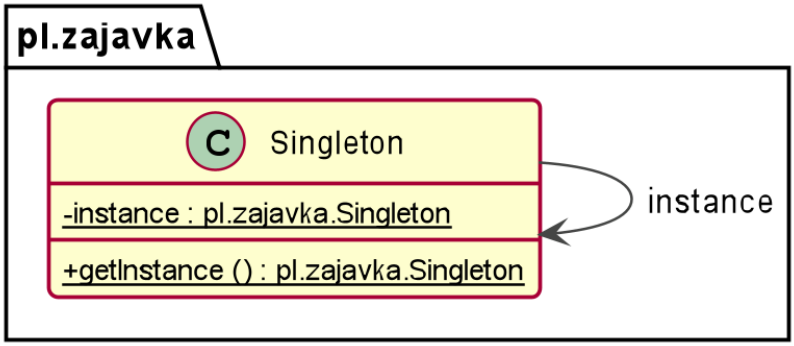
A co jeżeli nie chcemy tworzyć tego obiektu do momentu pierwszego wywołania metody `getInstance()`?

```
public final class LazyInitializationSingleton {

    private static LazyInitializationSingleton instance;

    private LazyInitializationSingleton() {}

    public static LazyInitializationSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializationSingleton();
        }
        return instance;
    }
}
```

Powyższa implementacja sprawdzi się w środowisku jednowątkowym (czyli takim jakiego dotychczas używamy cały czas). Jeżeli natomiast przyjdzie nam pisać w środowisku wielowątkowym, przyda nam się **thread-safe** `Singleton`, o którym powiemy sobie gdy przejdziemy do omawiania programowania wielowątkowego. Jeżeli zastosowalibyśmy pokazaną implementację `Singleton` w środowisku wielowątkowym, mogłoby to spowodować powstanie trudnych do przewidzenia i znalezienia błędów, np. mogłyby nam się utworzyć 2 instancje Singletona, ale nie jest to temat na teraz.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie:



Obraz 2. UML Singleton Class Diagram

Factory

Problem jaki rozwiązuje

Wzorzec projektowy **Fabryka** jest stosowany gdy potrzebujemy w elegancki sposób utworzyć instancję klasy, w sytuacji gdy mamy klasę bazową oraz kilka klas dziedziczących z niej. Na podstawie przekazanego parametru musimy stworzyć instancję jednej z klas dziedziczących z rodzica.

Przykład z życia

Wyobraź sobie, że mamy interface `Pizza` i w zależności od przekazanych argumentów chcemy zwrócić inną pizzę 😉.

Rozwiązanie

Rozwiązaniem jest stworzyć metodę, której typem zwracanym będzie klasa bazowa albo interfejs. W ciele tej metody, w zależności od przekazanego do niej argumentu wybieramy instancję obiektu dziedziczącego z klasy bazowej, który ma zostać utworzony.

Przykład w kodzie

Interface Pizza

```
public interface Pizza {
    String whatSauce();

    void bake();
}
```

Subklasa Pepperoni

```
public class PepperoniPizza implements Pizza {

    @Override
    public String whatSauce() {
        return "Spicy";
    }

    @Override
    public void bake() {
        System.out.println("I'm soo baking!");
    }

    public void beMoreSpicy() {
        System.out.println("Be prepared!");
    }
}
```

Subklasa Hawajska

```
public class HawaiianPizza implements Pizza {

    @Override
    public String whatSauce() {
        return "Tomato";
    }

    @Override
    public void bake() {
        System.out.println("I'm baking my pineapple!");
    }

    public void morePineapple() {
        System.out.println("More pineapple!");
    }
}
```

Fabryka

```
public class PizzaFactory {

    public static Pizza preparePizza(final String whatPizza) {
        switch (whatPizza) {
            case "Pepperoni":
                return new PepperoniPizza();
            case "Hawaiian":
                return new HawaiianPizza();
            default:
                throw new RuntimeException("I'm sorry, there is only Pepperoni or Hawaiian!");
        }
    }

}
```

PizzaRunner

```
public class PizzaRunner {

    public static void main(String[] args) {
        Pizza pepperoni = PizzaFactory.preparePizza("Pepperoni");
        Pizza hawaiian = PizzaFactory.preparePizza("Hawaiian");
    }
}
```

Zaznaczę tutaj, że to czy używamy `switch` w wersji oryginalnej, czy w wersji wprowadzonej w nowszych Javach, czy zamiast `switch` zastosujemy np. `Map<String, Supplier<Pizza>>`, gdzie kluczem jest nazwa pizzy podana jako `String`, a wartością implementacja interfejsu funkcyjnego `Supplier`, która dostarczy nam gotowy obiekt konkretnej pizzy nie ma znaczenia. Z resztą sposób z `Supplier` będzie pokazany za moment. Ważny tutaj jest sposób myślenia i samo podejście, a nie konstrukcja "mechanizmu wybierania" w środku.

Zalety tego podejścia:

- po wyjściu z fabryki operujemy na interfejsach, więc wspieramy podejście luźnego łączenia ze sobą klas,
- klient (czyli klasa, która faktycznie używa fabryki) jest nieświadoma implementacji pod spodem, nie interesuje jej w zasadzie co się tam dzieje, ona chce tylko pizzę. Dzięki temu osiągamy elastyczność polegającą na tym, że jeżeli chcemy dodać kolejny rodzaj pizzy do menu, to nie musimy nic zmieniać w klasie `FactoryRunner` (czyli w klasie korzystającej z fabryki), wystarczy, że dodamy kolejną opcję w samej fabryce,
- pokazane podejście zapewnia nam warstwę abstrakcji, a jak już zostało pokazane wcześniej - lepiej jest operować na abstrakcjach niż na konkretnych implementacjach.

Dokładamy Java 8

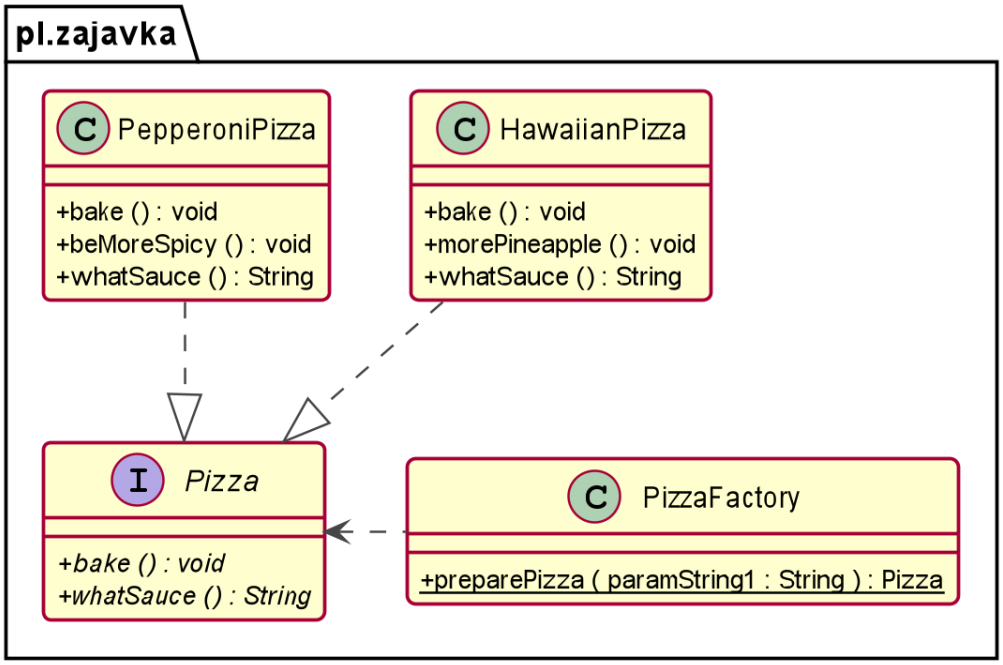
Patrząc na przykłady wyżej, można zauważyć, że implementacja tego wzorca jest możliwa przy wykorzystaniu lambdy (lub method reference) i zastosowaniu lekko innego podejścia:

```
public class LambdaPizzaFactory {

    final static Map<String, Supplier<Pizza>> PIZZA_MAP = new HashMap<>();

    static {
        PIZZA_MAP.put("Pepperoni", PepperoniPizza::new);
        PIZZA_MAP.put("Hawaiian", HawaiianPizza::new);
    }

    public static Pizza preparePizza(String pizzaType){
        return Optional.of(PIZZA_MAP.get(pizzaType))
            .map(Supplier::get)
            .orElseThrow(() -> new RuntimeException(
                String.format("I'm sorry, we don't know what [%s] is!", pizzaType)));
    }
}
```

Wywołanie metody z klasy `LambdaPizzaFactory` będzie wyglądało analogicznie do tego pokazanego w klasie `PizzaRunner`.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie:



Obraz 3. UML Factory Class Diagram

Abstract Factory

Fabryka abstrakcyjna (jakby to co dotychczas zostało pokazane nie było wystarczająco abstrakcyjne 😁). Wzorzec ten jest bardzo podobny do poprzedniego, ale jest bardziej abstrakcyjny. Można go rozumieć jako fabrykę fabryk.

Problem jaki rozwiązuje

W przypadku zwykłej fabryki, mieliśmy jedną klasę, w której metoda zwracała subklasę bazując na parametrze będącym argumentem metody. Fabrykę abstrakcyjną zastosujemy natomiast gdy będziemy chcieli uzyskać metodę, która zwróci nam rodzinę zależnych obiektów bez określana jakie konkretnie obiekty to są. Mówiąc rodzina obiektów można rozumieć to jako konkretną fabrykę danego typu obiektu, która może taki obiekt wyprodukować. Zaraz wyjaśnimy to sobie w kodzie.

Przykład z życia

Nadal będziemy obracać się w przykładzie Pizzy, ale skoro fabryka abstrakcyjna jest w stanie stworzyć fabrykę fabryk, to w tym przykładzie stworzymy fabrykę pizz oraz fabrykę samochodów.

Rozwiązanie

Korzystając z fabryki jaką poznaliśmy w poprzednim przykładzie, dodajemy jeszcze klasę fabryki abstrakcyjnej, która zwróci nam naszą konkretną fabrykę przy spełnieniu określonych warunków.

Przykład w kodzie

Interface Pizza

```
public interface Pizza {
    String whatSauce();

    void bake();
}
```

Subklasa Pepperoni

```
public class PepperoniPizza implements Pizza {

    @Override
    public String whatSauce() {
        return "Spicy";
    }

    @Override
    public void bake() {
        System.out.println("I'm soo baking!");
    }

    public void beMoreSpicy() {
        System.out.println("Be prepared!");
    }
}
```

Subklasa Hawajska

```
public class HawaiianPizza implements Pizza {

    @Override
    public String whatSauce() {
        return "Tomato";
    }

    @Override
    public void bake() {
        System.out.println("I'm baking my pineapple!");
    }

    public void morePineapple() {
        System.out.println("More pineapple!");
    }
}
```

Teraz stworzymy interfejs fabryki abstrakcyjnej, który będzie następnie wykorzystany przez konkretne

fabryki: fabrykę dla interfejsu `Pizza` i fabrykę dla interfejsu `Car`. Struktura interfejsu `Car` i klas pochodnych jest analogiczna jak struktura interfejsu `Pizza` i jego klas pochodnych.

Interface Fabryki Abstrakcyjnej AbstractFactory

```
public interface AbstractFactory<T> {

    T create(String type) ;
}
```

Zwróć uwagę, że metoda `create()` zwraca typ generyczny. Teraz musimy stworzyć klasę fabrykę, która zaimplementuje ten interface i zwróci konkretne subklasy.

Fabryka PizzaFactory

```
public class PizzaFactory implements AbstractFactory<Pizza> {

    @Override
    public Pizza create(final String type) {
        switch (type) {
            case "Pepperoni":
                return new PepperoniPizza();
            case "Hawaiian":
                return new HawaiianPizza();
            default:
                throw new RuntimeException("I'm sorry, there is only Pepperoni or Hawaiian!");
        }
    }
}
```

Do tego tworzymy analogiczną klasę `CarFactory`, której nie pokazuję w przykładzie, żeby ograniczyć ilość powielanego kodu.

Mając już utworzone takie klasy i interfejsy, możemy stworzyć klasę `FactoryProvider`.

Klasa FactoryProvider

```
public class FactoryProvider {

    public static AbstractFactory<?> getFactory(String whatFactory) {
        switch (whatFactory) {
            case "Pizza":
                return new PizzaFactory();
            case "Car":
                return new CarFactory();
            default:
                throw new RuntimeException("I'm sorry, we only produce pizzas or cars!");
        }
    }

}
```

Klasa `FactoryProvider`, w metodzie `getFactory()` przyjmuje jako argument informację o tym jaka fabryka ma zostać zwrócona. Metoda `getFactory()` jak sama nazwa wskazuje - zwraca fabrykę.

Zauważ też stosowany **unbounded wildcard**, który powoduje, że w przykładzie poniżej musimy przypisać wynik wywołania `FactoryProvider` do klasy `Object`. Możemy oczywiście pokusić się w tym miejscu o rzutowanie, ale nie dodaję tego w przykładzie.

Możemy teraz użyć powyższej konstrukcji w ten sposób.

```
public class FactoryRunner {

    public static void main(String[] args) {
        Object o = FactoryProvider.getFactory("Pizza").create("Hawaiian");
        System.out.println(o);
    }

}
```

Zalety tego podejścia:

- klient (czyli klasa `FactoryRunner`) jest niezależna od sposobu tworzenia fabryki, zmiana w implementacji fabryki abstrakcyjnej nie wpływa na zmianę kodu klasy `FactoryRunner`,
- stosując to podejście, możemy w trakcie działania programu zdecydować jaką fabrykę utworzymy - w trakcie działania programu, bo decydujemy o tym przy pomocy parametru `String`, a nie na etapie kompilacji,
- bez ingerencji w istniejące fabryki możemy dodać logikę odpowiedzialną za tworzenie innej rodziny obiektów, np. komputerów.

Dokładamy Java 8

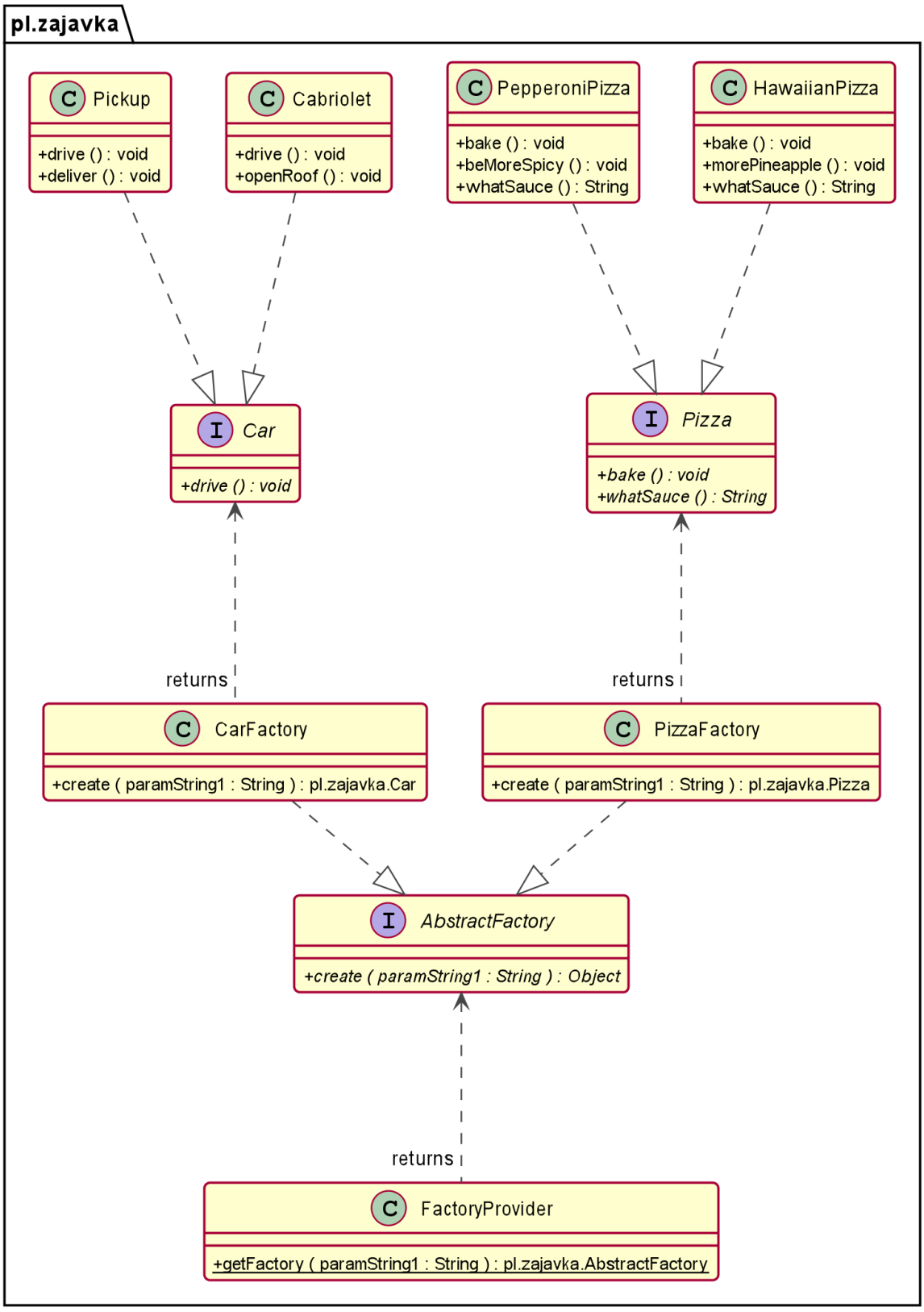
Tak samo jak w przypadku wzorca **Factory**, również **Abstract Factory** może zostać zaimplementowane przy wykorzystaniu lambdy (lub method reference). Jedyne co trzeba zmienić to implementacja klasy `FactoryProvider` na poniższą:

```
public class LambdaFactoryProvider {

    final static Map<String, Supplier<AbstractFactory<?>>> FACTORIES_MAP = new HashMap<>();

    static {
        FACTORIES_MAP.put("Pizza", PizzaFactory::new);
        FACTORIES_MAP.put("Car", CarFactory::new);
    }

    public static AbstractFactory<?> getFactory(String whatFactory){
        return Optional.of(FACTORIES_MAP.get(whatFactory))
            .map(Supplier::get)
            .orElseThrow(() -> new RuntimeException("I'm sorry, we only produce pizzas or cars!"));
    }

}
```

Jednocześnie też możemy wymienić implementację poszczególnych fabryk na te, które korzystają z podejścia funkcyjnego.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie:



Obraz 4. UML Abstract Factory Class Diagram

Builder

Jeżeli razem z nami uczyłeś/uczyłaś się o `Lombok`, to pamiętasz może adnotację `@Builder`? A jeżeli nie to zakładam, że wiesz już czym jest adnotacja `@Builder` w `Lombok`. Teraz dowiemy się trochę o filozofii jaka za tym stoi.

Problem jaki rozwiązuje

Jeżeli mamy klasę (np. `Car`), która składa się z bardzo dużej ilości pól i chcemy inicjalizować dużą ilość tych pól podczas tworzenia obiektu tej klasy (`Car`) to pojawi się problem, jeżeli za każdym razem będziemy potrzebowali inicjować inne pola. **Builder** służy do tego, aby móc za każdym razem na etapie tworzenia obiektu wybrać pola, które mają zostać zainicjowane.

Bez tego wzorca musielibyśmy tworzyć różne kombinacje konstruktorów z różnymi listami parametrów aby rozwiązać ten problem. Ewentualnie tworzyć obiekt i ustawiać wartości tych pól przy wykorzystaniu setterów, ale zapis taki nie jest już taki zwięzły. Konstruktory mają też ten problem, że jeżeli będziemy przekazywali 10 argumentów typu `String` to bardzo łatwo można się pomylić, który argument powinien być na którym miejscu. W wariancie z setterami natomiast, stan obiektu zmienia się przez całą ścieżkę wywoływania setterów. Nie rozwiąże to problemu, w którym chcielibyśmy mieć zainicjowane wartości pól na etapie tworzenia obiektu i obiekt **immutable**. Settery zmieniają stan po faktycznym utworzeniu obiektu.

Przykład z życia

Samochód jest dobrym przykładem klasy, która może składać się z bardzo dużej ilości pól i jednocześnie wiele z tych pól jest opcjonalne. Przecież nie każdy samochód ma hak holowniczy, szyberdach czy bagażnik na narty. Jeżeli natomiast definiowalibyśmy taką klasę w kodzie, to przecież musimy przygotować miejsce na te wartości.

Rozwiązanie

Na rozwiązanie składa się kilka kroków:

- w pierwszym kroku utworzymy statyczną klasę zagnieżdżoną, która będzie miała w nazwie `Builder`, czyli będzie to np. `CarBuilder`. Następnie skopiujemy do tej klasy wszystkie pola, które są zdefiniowane w klasie, której obiekt będziemy budować, np. `Car`.
- klasa `Car` powinna mieć statyczną metodę `builder()`, która tworzy instancję klasy `CarBuilder`.
- klasa `CarBuilder` powinna mieć zdefiniowane metody, które ustawią wartości konkretnych parametrów, a następnie zwrócą instancję klasy `CarBuilder`.
- w klasie `Car` definiujemy prywatny konstruktor, który jako argument przyjmuje klasę `CarBuilder`, a następnie inicjuje pola na podstawie wartości pól z klasy `CarBuilder.`
- w Klasie `CarBuilder` definiujemy metodę `build()`, która wywoła konstruktor prywatny z klasy `Car`.

Przykład w kodzie

Klasa Car

```
public class Car {

    private final String brand;
    private final String model;
    private final Integer year;
    private final String towbar;
    private final String sunRoof;
    private final String skiRack;

    private Car(final CarBuilder carBuilder) {
        this.brand = carBuilder.brand;
        this.model = carBuilder.model;
        this.year = carBuilder.year;
        this.towbar = carBuilder.towbar;
        this.sunRoof = carBuilder.sunRoof;
        this.skiRack = carBuilder.skiRack;
    }

    // gettery

    public static CarBuilder builder() {
        return new CarBuilder();
    }

    public static class CarBuilder {
        private String brand;
        private String model;
        private Integer year;
        private String towbar;
        private String sunRoof;
        private String skiRack;

        public CarBuilder brand(final String brand) {
            this.brand = brand;
            return this;
        }

        public CarBuilder model(final String model) {
            this.model = model;
            return this;
        }

        public CarBuilder year(final Integer year) {
            this.year = year;
            return this;
        }

        public CarBuilder towbar(final String towbar) {
            this.towbar = towbar;
            return this;
        }

        public CarBuilder sunRoof(final String sunRoof) {
            this.sunRoof = sunRoof;
            return this;
        }
```

```
        public CarBuilder skiRack(final String skiRack) {
            this.skiRack = skiRack;
            return this;
        }

        public Car build() {
            return new Car(this);
        }
    }
}
```

Użycie Buildera klasy Car

```
public class CarCreator {

    void create() {
        Car emptyCar = Car.builder().build();

        Car anotherCar = Car.builder()
            .brand("Ford")
            .model("Mustang")
            .towbar("someTowbar")
            .year(1969)
            .build();
    }
}
```

Jeżeli teraz chcielibyśmy wydrukować zawartość obiektu pod referencją `anotherCar` na ekranie (po dodaniu metody `toString()`), wyglądałoby to tak:

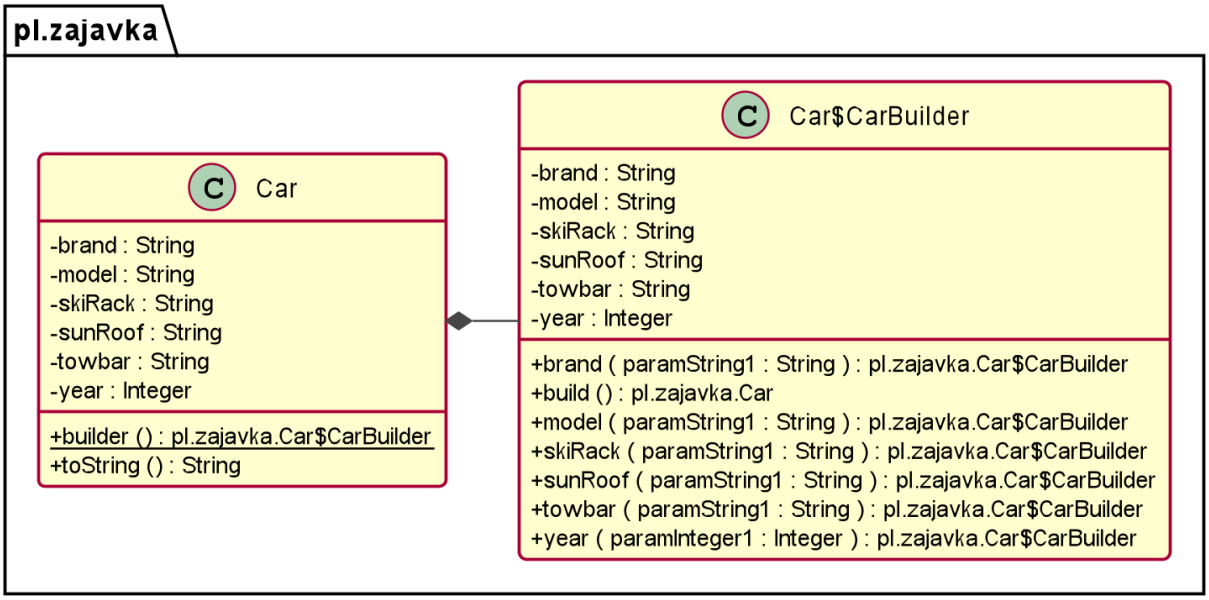
```
Car{brand='Ford', model='Mustang', year=1969, towbar='someTowbar', sunRoof='null', skiRack='null'}
```

Widzisz zatem, że możesz teraz tworzyć różne samochody w różnych konfiguracjach i jednocześnie zapisać to w bardzo przystępny sposób.

W praktyce tego wzorca używa się bardzo często, dlatego też warto jest pamiętać o adnotacji `@Builder`, która przychodzi razem z **Lombok**. Dzięki niej nie musimy pisać tego wzorca sami! Zobacz również z ciekawości jak będzie wyglądał ten wzorzec jeżeli zastosujesz na klasie `Car` adnotację `@Builder` i uruchomisz **delombok** 😄.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie:



Obraz 5. UML Builder Class Diagram

Przypomnę, że `$` jest wykorzystywany przez Javę, do oznaczenia skompilowanego pliku `.class` z klasą zagnieżdżoną. Jednocześnie na diagramie relacja pomiędzy tymi klasami została przedstawiona w formie kompozycji, gdyż to klasa `Car` decyduje o życiu klasy `CarBuilder`. Przykład ten jednocześnie pokazuje, że **UML** nie został stworzony do tego, żeby odwzorowywać mechanizmy Javy 1:1. Jeżeli trafimy na język programowania, w którym nie występuje koncepcja klas zagnieżdżonych to ten sam diagram **UML** byłby reprezentowany w inny sposób. Z drugiej strony patrząc na taki diagram, trzeba wiedzieć, że `$` oznacza w Javie klasę zagnieżdżoną, bo inaczej możemy zrozumieć, że klasa `CarBuilder` jest polem w klasie `Car`. Zwracam uwagę na takie niuanse, żeby poszerzać Twoją świadomość jako programisty/programistki ☺.

Lombok

I wyobraź sobie teraz, że zamiast pisać tego wszystkiego wystarczy użyć adnotacji `@Builder`, którą omawialiśmy w ramach warsztatu o `Lomboku`.

Prototype

Ostatnim wzorcem kreacyjnym który poruszymy jest **Prototype**.

Problem jaki rozwiązuje

Prototype jest stosowany gdy potrzebujemy stworzyć obiekt, którego stworzenie jest kosztowne w rozumieniu czasu i zasobów. Jednocześnie możemy skorzystać z tego, że podobny obiekt już istnieje.

Przykład z życia

Nazwa jest o tyle niefortunna, że prototyp w języku codziennym oznacza wersję produktu, która jest testowana zanim powstanie wersja finalna. W przypadku tego wzorca, prototyp jest źródłem, z którego powstaje kopia. Lepszym przykładem byłoby tutaj spisywanie pracy domowej przed lekcją 😄.

Przepisujesz czyjeś zadanie, inaczej mówiąc klonujesz czyjeś zadanie i zmieniasz parę rzeczy mówiąc, że zostało one rozwiązane przez Ciebie samodzielnie.

Rozwiązanie

Prototype daje nam możliwość wykonania kopii obiektu już istniejącego i modyfikacji wartości, które powinny być zmodyfikowane. Stosując ten wzorzec projektowy utworzymy nowy obiekt na podstawie już istniejącego (czyli skopiujemy wartości) i zmienimy te wartości, które nas interesują. Kod, który będzie wykonywał kopiowanie umieszczamy w klasie, która ma być kopiowana. Podejście to przekłada się na to, że mamy dostęp do pól prywatnych obiektu, który chcemy skopiować.

Przykład w kodzie

Wzorzec ten może zostać zaimplementowany przy wykorzystaniu interfejsu `Cloneable` i metody `clone()`, która pozwala nam stworzyć kopię obiektu.

Zanim przejdziemy dalej, chciałbym jednak wyjaśnić 2 pojęcia, **shallow copy** i **deep copy**. Jeżeli przyjdzie nam kiedyś rozmawiać o kopiowaniu obiektów, możemy zrobić to albo płytko, albo głęboko.

Płytka kopia nastąpi wtedy, gdy utworzymy nowy obiekt, ale będziemy wskazywać na te same referencje, na który wskazywał obiekt źródło. **Kopia głęboka** wystąpi natomiast wtedy gdy zarezerwujemy nowe miejsce w pamięci dla każdego pola zagnieżdżonego (`Car` > `SteeringWheel` > `Color` itp.), czyli stworzymy każdy z takich obiektów ponownie, tylko przy pomocy starych wartości. Teoretycznie w takim przypadku zużyjemy 2 razy więcej pamięci. Poniżej przykład w kodzie:

Klasa CopyCar

```
@Data
public class CopyCar {

    SteeringWheel steeringWheel;

    public CopyCar shallowCopy() {
        var newCar = new CopyCar();
        newCar.steeringWheel = this.steeringWheel;
        return newCar;
    }

    public CopyCar deepCopy() {
        var newCar = new CopyCar();
        newCar.steeringWheel = this.steeringWheel.deepCopy();
        return newCar;
    }

}
```

Klasa SteeringWheel

```
@Data
@NoArgsConstructor
@AllArgsConstructor(staticName = "of")
class SteeringWheel {
    private double diameter;
```

```
    public SteeringWheel deepCopy() {
        var newSteeringWheel = new SteeringWheel();
        newSteeringWheel.diameter = this.diameter;
        return newSteeringWheel;
    }
}
```

Klasa CopyCarRunner

```
public class CopyCarRunner {

    public static void main(String[] args) {
        CopyCar car = new CopyCar();
        SteeringWheel steeringWheel = new SteeringWheel();
        steeringWheel.setDiameter(21.5);
        car.setSteeringWheel(steeringWheel);

        CopyCar shallowCopied = car.shallowCopy();
        CopyCar deepCopied = car.deepCopy();

        System.out.println("shallowCopied == "
            + (car.steeringWheel == shallowCopied.steeringWheel)); ①
        System.out.println("shallowCopied equals "
            + (car.steeringWheel.equals(shallowCopied.steeringWheel))); ②
        System.out.println("deepCopied == "
            + (car.steeringWheel == deepCopied.steeringWheel)); ③
        System.out.println("deepCopied equals "
            + (car.steeringWheel.equals(deepCopied.steeringWheel))); ④
    }
}
```

① Zostanie wydrukowane: *shallowCopied* == *true*, ponieważ **płytka** kopia wskazuje na to samo miejsce w pamięci.

② Zostanie wydrukowane: *shallowCopied equals true*, ponieważ metoda `equals()` została zaimplementowana dzięki adnotacji `@Data`, a wartości porównywane są takie same.

③ Zostanie wydrukowane: *deepCopied* == *false*, ponieważ **głęboka** kopia wskazuje na inne miejsce w pamięci.

④ Zostanie wydrukowane: *deepCopied equals true*, ponieważ metoda `equals()` została zaimplementowana dzięki adnotacji `@Data`, a wartości porównywane są takie same.

Wracając do przykładu Prototype. Jeżeli taka możliwość istniałaby w rzeczywistości, to producenci samochodów na pewno bardzo chętnie by z tego skorzystali. Nie trzeba by wtedy było produkować samochodu od zera, tylko wykonalibyśmy kopię poprzedniego, pomalowali na inny kolor i sprzedane 😄.

Przykład klasy Car

```
@Data
public class PrototypeCar implements Cloneable {

    private String brand;
    private String model;
    private SteeringWheel steeringWheel;
    private List<String> doors;
```

```
    public PrototypeCar() {
        this.brand = "Ford";
        this.model = "Mustang";
        this.steeringWheel = SteeringWheel.of(12.34);
        // załóżmy, że tworzenie drzwi jest bardzo kosztowne czasowo
        this.doors = new ArrayList<>(
            Arrays.asList("left front", "right front", "left rear", "right rear"));
    }

    @Override
    protected PrototypeCar clone() throws CloneNotSupportedException {
        final PrototypeCar clone = (PrototypeCar) super.clone();
        clone.brand = this.brand;
        clone.model = this.model;
        clone.steeringWheel = this.steeringWheel.clone();
        clone.doors = new ArrayList<>(this.doors);
        return clone;
    }
}
```

Dodam w tym miejscu, że jeżeli metodę `clone()` zaimplementowalibyśmy w sposób jak poniżej to otrzymalibyśmy **płytką** kopię obiektu:

```
@Override
protected PrototypeCar clone() {
    return super.clone();
}
```

Wracając do przykładu:

Przykład klasy SteeringWheel

```
@Data
@NoArgsConstructor
@AllArgsConstructor(staticName = "of")
class SteeringWheel implements Cloneable {
    private double diameter;

    @Override
    protected SteeringWheel clone() throws CloneNotSupportedException {
        final SteeringWheel clone = (SteeringWheel) super.clone();
        clone.diameter = this.diameter;
        return clone;
    }
}
```

I klasa, która uruchamia powyższy kod

```
public class PrototypeCarRunner {

    @SneakyThrows
    public static void main(String[] args) {
        // załóżmy, że ta operacja jest kosztowna
        PrototypeCar prototypeCar = new PrototypeCar(); ①
        System.out.println("original: " + prototypeCar);
```

```
        PrototypeCar cloned = prototypeCar.clone();
        System.out.println("cloned: " + cloned);

        System.out.println(prototypeCar.getSteeringWheel() == cloned.getSteeringWheel());
        System.out.println(prototypeCar.getBrand() == cloned.getBrand());
        System.out.println(prototypeCar.getModel() == cloned.getModel()); ②
        System.out.println(prototypeCar.getDoors() == cloned.getDoors());

        System.out.println("before adding: " + cloned);
        cloned.getDoors().add("back door");

        System.out.println("after adding proto: " + prototypeCar);
        System.out.println("after adding cloned: " + cloned); ③
    }
}
```

① Załóżmy, że ta operacja jest kosztowna.

② Porównanie Stringów zwróci `true` ze względu na **String pool**.

③ Nowe drzwi zostały dołożone tylko do sklonowanego samochodu.

W powyższym przykładzie wykonaliśmy **deep copy** obiektu klasy `PrototypeCar` i dodaliśmy do obiektu `cloned` tylne drzwi. Jeżeli uruchomisz ten kod to zobaczysz, że nowe drzwi zostały dodane tylko do klona, oryginalny obiekt pozostał taki jaki był.

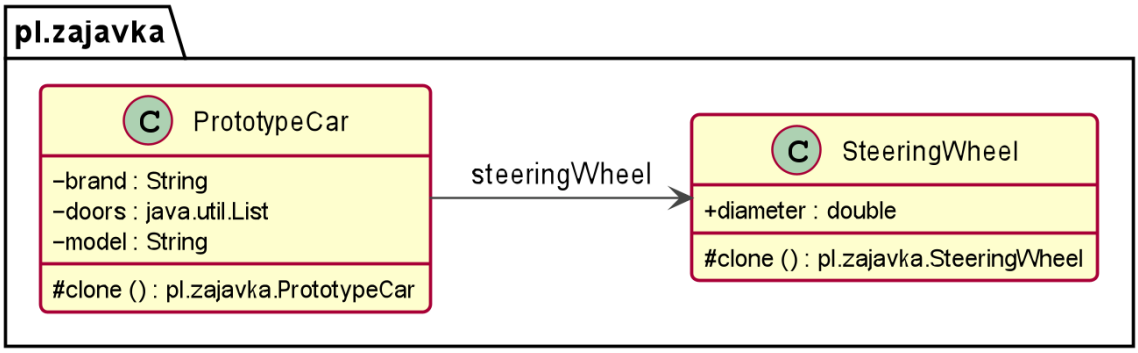
W przykładach wykorzystywaliśmy również metodę `super.clone()`, gdyż dokumentacja klasy Object wspomina o takiej konwencji klonowania obiektów przy wykorzystaniu tej metody.

Zalety tego podejścia:

- Wzorzec **Prototype** pozwala skorzystać z istniejących już obiektów w celu oszczędzenia czasu przy tworzeniu nowego złożonego obiektu,
- Podejście to jest bardzo przydatne gdy chcemy zapisywać historię aktualizacji obiektu. Tworzymy wtedy kopię istniejącego już obiektu i zmieniamy tylko te wartości, które zostały zaktualizowane,
- Stosujemy zasadę **DRY**, czyli pozbywamy się zduplikowanego kodu, który mógłby służyć do tworzenia różnych wariantów tego samego obiektu,
- Tworzenie bardzo złożonych obiektów jest o wiele prostsze i wygodniejsze.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie:



Obraz 6. UML Prototype Class Diagram

Lombok

@With

Pamiętasz może adnotację `@With`, która była omawiana w ramach warsztatu o `Lomboku`? Czy zachowanie tej adnotacji nie przypomina Ci czegoś 😁? Spójrz na poniższy przykład wykorzystujący adnotację `@With`.

Klasa LombokPrototypeCar

```
@Data
@With
@AllArgsConstructor
public class LombokPrototypeCar {

    private String brand;
    private String model;
    private LombokSteeringWheel steeringWheel;
    private List<String> doors;

    public LombokPrototypeCar() {
        this.brand = "Ford";
        this.model = "Mustang";
        this.steeringWheel = new LombokSteeringWheel(12.34);
        this.doors = new ArrayList<>(
            Arrays.asList("left front", "right front", "left rear", "right rear"));
    }
}
```

Klasa LombokSteeringWheel

```
@Data
@With
@NoArgsConstructor
@AllArgsConstructor
class LombokSteeringWheel {
    private double diameter;
}
```

Klasa LombokPrototypeCarRunner

```
public class LombokPrototypeCarRunner {
```

```
    @SneakyThrows
    public static void main(String[] args) {
        LombokPrototypeCar origin = new LombokPrototypeCar();
        System.out.println("origin before: " + origin);

        List<String> newDoors = new ArrayList<>(origin.getDoors());
        newDoors.add("back door");
        LombokPrototypeCar cloned = origin.withDoors(newDoors); ①

        System.out.println("origin after: " + origin);
        System.out.println("arigin cloned: " + cloned); ②

        // sprawdźmy co się teraz stanie
        System.out.println("car == " + (origin == cloned)); ③
        System.out.println("car equals " + (origin.equals(cloned))); ④
        System.out.println("steeringWheel == "
            + (origin.getSteeringWheel() == cloned.getSteeringWheel())); ⑤
        System.out.println("steeringWheel equals "
            + (origin.getSteeringWheel().equals(cloned.getSteeringWheel()))); ⑥
    }
}
```

① Wykorzystanie adnotacji `@With`.

② Zwróć uwagę, że oryginalny obiekt nie został zmodyfikowany.

③ Na ekranie zostanie wydrukowane: *car == false*.

④ Na ekranie zostanie wydrukowane: *car equals false*.

⑤ Na ekranie zostanie wydrukowane: *steeringWheel == true*.

⑥ Na ekranie zostanie wydrukowane: *steeringWheel equals true*.

Powyższe zachowanie jasno pokazuje, że adnotacja `@With` stosuje **shallow copy**, gdyż:

- Dostajemy nowy obiekt `LombokPrototypeCar`, co pokazuje nam wiadomość: *car == false*,
- Obiekt `LombokSteeringWheel` nadal jest tym samym obiektem (to samo miejsce w pamięci), co pokazuje nam wiadomość: *steeringWheel == true*.

Jaki można wyciągnąć wniosek patrząc na przykład z `@With`? Jeżeli chcielibyśmy wykorzystać wzorzec **prototype** dla obiektu, który posiada tylko pola prymitywne, albo pola są klasami immutable (czyli i tak stan obiektów tych klas nigdy nie będzie ulegał zmianie) to spokojnie możemy robić **shallow copy** i wykorzystywać w tym celu adnotację `@With`. Jeżeli natomiast będzie zależało nam na zrobieniu **deep copy**, czyli żeby obiekty zdefiniowane jako pola również były tworzone na nowo i zajmowały nowe miejsce w pamięci, to musielibyśmy zastosować inną implementację i nie możemy wtedy polegać na `@With`. Chyba, że **shallow copy** jest dla nas w zupełności wystarczające.

@Builder

Skoro jesteśmy już znowu przy `Lomboku` to warto jest tutaj poruszyć jeszcze jedną kwestię. `Lombok` daje nam możliwość zapisania adnotacji `@Builder` w ten sposób:

```
@Data
@With
@Builder(toBuilder = true)
```

```
@AllArgsConstructor
public class LombokPrototypeCar {
    // ...
}
```

Wykorzystanie tej adnotacji daje nam możliwość zapisania kodu w poniższy sposób:

```
public class LombokPrototypeCarRunner {

    @SneakyThrows
    public static void main(String[] args) {
        LombokPrototypeCar origin = new LombokPrototypeCar();

        System.out.println("origin: " + origin);
        List<String> newDoors = new ArrayList<>(origin.getDoors());
        newDoors.add("back door");
        LombokPrototypeCar cloned = origin.toBuilder() ①
            .doors(newDoors)
            .build();

        System.out.println("after adding origin: " + origin);
        System.out.println("after adding cloned: " + cloned); ②

        // sprawdźmy co się teraz stanie
        System.out.println("== " + (origin == cloned)); ③
        System.out.println("equals " + (origin.equals(cloned))); ④
        System.out.println("steeringWheel == "
            + (origin.getSteeringWheel() == cloned.getSteeringWheel())); ⑤
        System.out.println("steeringWheel equals "
            + (origin.getSteeringWheel().equals(cloned.getSteeringWheel()))); ⑥
    }
}
```

① Wykorzystanie adnotacji `@Builder(toBuilder = true)`.

② Zwróć uwagę, że oryginalny obiekt nie został zmodyfikowany.

③ Na ekranie zostanie wydrukowane: *car == false*.

④ Na ekranie zostanie wydrukowane: *car equals false*.

⑤ Na ekranie zostanie wydrukowane: *steeringWheel == true*.

⑥ Na ekranie zostanie wydrukowane: *steeringWheel equals true*.

Ponownie zostało zrobione **shallow copy**, widać to analogicznie jak w poprzednim przykładzie.

Podsumowanie

Grupa kreacyjnych wzorców projektowych skupia się wokół tworzenia obiektów. Pokazane zostały sposoby inicjalizacji obiektów albo grup obiektów w sposób uniwersalny, znany w świecie programowania. Warto pamiętać o tych wzorcach gdy będziemy chcieli zachować czystą strukturę naszego kodu.