Design Patterns

Spis treści

Behavioral Design Patterns

W przypadku wzorców **behawioralnych** nacisk kładziony jest na zapewnienie rozwiązań, które pozwalają nam w luźny sposób powiązać ze sobą obiekty między którymi następuje interakcja w kodzie. Dążymy tutaj do elastyczności tych interakcji po to aby każdy z tych obiektów mógł być łatwo rozszerzalny bez wpływu na inne obiekty pozostające w interakcji. Przykładowe wzorce **behawioralne** to (wyjaśniamy je po kolei poniżej):

- Template method
- Chain of Responsibility
- Observer
- Strategy
- Visitor

Przejdźmy teraz do omówienia każdego z nich.

Template method

Problem jaki rozwiązuje

Wzorzec **Template method** jest używany gdy chcemy określić jedno miejsce w kodzie, które w abstrakcyjny sposób określi kroki naszego algorytmu, natomiast konkretna implementacja tych kroków zostanie oddelegowana do konkretnych klas implementujących. Jednocześnie możemy zapewnić domyślną implementację takich kroków, która może być nadpisana w klasach implementujących konkretne kroki.

Przykład z życia

Dobrym przykładem z życia jest sytuacja, w której mamy konkretne kroki do wykonania i chcemy je spisać w abstrakcyjnej formie. Sytuacją taką może być budowa domu. Algorytm budowy domu składa się z takich abstrakcyjnych kroków jak zrobienie wykopu, zalanie fundamentów, postawienie ścian, nakrycie dachem, wstawienie okien, zrobienie elewacji itp. Widać już, że budowa domu składa się z abstrakcyjnych kroków, którym należy zapewnić konkretną implementację. Ważne też jest to, że często nie możemy zmienić kolejności tych kroków, tzn. wstawić okien przed postawieniem ścian. Różne domy, w zależności od tego czy są drewniane czy murowane będą miały inne implementacje konkretnych kroków.

Rozwiązanie

Rozwiązanie będzie polegało na stworzeniu klasy abstrakcyjnej, która określa kolejne kroki jakie muszą zostać wykonane aby wybudować konkretny dom. Różne domy natomiast, w zależności od tego, czy są drewniane czy murowane będą różniły się konkretnymi implementacjami tych metod. Czyli inaczej wstawimy okna w domu drewnianym, a inaczej w domu murowanym. Mając klasę abstrakcyjną, która określa konkretne kroki do wykonania, dodajemy jeszcze konkretne klasy takie jak np. `WoodenHouse` i `BrickHouse`. Metodę, która określa schemat wykonania kolejnych metod możemy napisać jako **final** aby

zablokować możliwość jej nadpisania przez klasy dziedziczące. Można powiedzieć, że wykop pod fundamenty jest taki sam dla obu domów, więc możemy tutaj wykorzystać metodę domyślną.

Przykład w kodzie

Zacznijmy od stworzenia klasy abstrakcyjnej `HouseTemplate`, która ma zdefiniowaną template metodę - `build()`.

Klasa HouseTemplate

```
public abstract class HouseTemplate {

    protected final void build() {
        digHole();
        buildFoundation();
        buildWalls();
        makeRoof();
        insertWindows();
    }

    protected void digHole() {
        System.out.println("digging hole");
    }

    protected abstract void buildFoundation();

    protected abstract void buildWalls();

    protected abstract void makeRoof();

    protected abstract void insertWindows();
}
```

Klasa ma zdefiniowaną metodę `digHole()`, która ma zapewnioną implementację domyślną, która może być taka sama niezależnie od tego jaki stawiamy dom. Reszta metod (oprócz `build()`) jest abstrakcyjna, więc potrzebujemy klasy, która będzie dziedziczyła z klasy `HouseTemplate`, aby móc nadpisać te metody. W tym celu stwórzmy klasy `WoodenHouse` i `BrickHouse`, które będą określały inne implementacje metod abstrakcyjnych, które będą charakterystyczne dla danego przypadku domu.

Klasa WoodenHouse

```
public class WoodenHouse extends HouseTemplate {

    @Override
    protected void buildFoundation() {
        System.out.println("WoodenHouse buildFoundation");
    }

    @Override
    protected void buildWalls() {
        System.out.println("WoodenHouse buildWalls");
    }

    @Override
    protected void makeRoof() {
        System.out.println("WoodenHouse makeRoof");
```

```
    }

    @Override
    protected void insertWindows() {
        System.out.println("WoodenHouse insertWindows");
    }
}
```

Klasa BrickHouse

```
public class BrickHouse extends HouseTemplate {

    @Override
    protected void buildFoundation() {
        System.out.println("BrickHouse buildFoundation");
    }

    @Override
    protected void buildWalls() {
        System.out.println("BrickHouse buildWalls");
    }

    @Override
    protected void makeRoof() {
        System.out.println("BrickHouse makeRoof");
    }

    @Override
    protected void insertWindows() {
        System.out.println("BrickHouse insertWindows");
    }
}
```

Możemy teraz uruchomić ten kod z klasy `HouseRunner`.

Klasa HouseRunner

```
public class HouseRunner {

    public static void main(String[] args) {
        HouseTemplate house1 = new BrickHouse();
        house1.build();

        HouseTemplate house2 = new WoodenHouse();
        house2.build();
    }
}
```

Dzięki wykorzystaniu **polimorfizmu** możemy wykorzystać klasę `HouseTemplate` jako referencję, natomiast podstawić pod te referencje konkretne implementacje w postaci klas `BrickHouse` oraz `WoodenHouse`. W zależności od implementacji zostaną wywołane inne metody.

Zalety tego podejścia:

- Dzięki stosowaniu tego podejścia możemy określić sztywną listę kroków, która ma zostać zaimplementowana przez klasę implementującą.

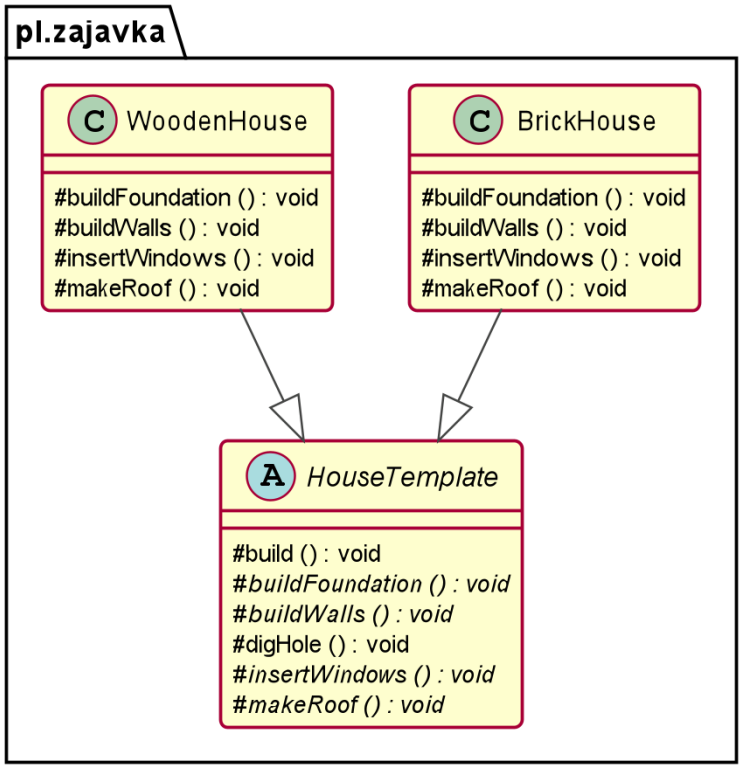
- Jeżeli chcemy aby któraś metoda nie była nadpisywana możemy oznaczyć ją jako final.

Na co zwrócić szczególną uwagę?

- Z racji, że wykorzystujemy klasę abstrakcyjną, to należy pamiętać, że każda modyfikacja tej klasy wpłynie na każdą klasę, która dziedziczy z klasy abstrakcyjnej `HouseTemplate`. Dlatego właśnie należy uważać na to, aby nie próbować dziedziczyć na siłę z jakiejś klasy bazowej.

Diagram UML

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



Obraz 1. UML Template Method Class Diagram

Chain of Responsibility

Problem jaki rozwiązuje

Wzorzec ten pozwala nam na zaprojektowanie łańcucha obiektów, które będą służyły do przetworzenia jakiegoś fragmentu logiki. Logika taka może być wtedy podzielona na kolejne kroki w łańcuchu przetwarzania. Każdy obiekt będący ogniwem takiego łańcucha może wtedy zdecydować czy kontynuować wykonywanie logiki, czy przerwać w danym momencie, ze względu na niespełnienie pewnych warunków. Wzorzec ten jest używany do osiągnięcia **luźnego powiązania** (*loose coupling*) między obiektami. Możemy stosować ten wzorzec jeżeli dopiero w trakcie działania programu jesteśmy w stanie określić który obiekt ma przetwarzać naszą "logikę" w kodzie. Równie dobrze może się okazać, że żaden obiekt nie będzie w stanie przetworzyć danych wejściowych, które chcemy przetworzyć.

Czyli upraszczając, możemy w ten sposób napisać łańcuch obiektów, które są w stanie obsłużyć prośbę obiektu zwanego **Klientem**. W dalszej części stanie się to bardziej jasne ☺.

Przykład z życia

Jako przykład z życia wyobraźmy sobie łańcuch stanowisk na których tworzony jest jakiś produkt. W takim przypadku możemy wyróżnić klienta, który składa żądanie wytworzenia jakiegoś produktu. Żądanie takie razem ze wszystkimi parametrami przechodzi przez wszystkie stanowiska, które w danej kolejności mogą procesować takie żądanie. Każde stanowisko może zidentyfikować czy jest w stanie przetworzyć fragment żądania, jeżeli tak - przetwarza je i w następnej kolejności przekazuje to samo żądanie do następnego stanowiska. Dla prostoty opisany zostanie bardzo uproszczony łańcuch tworzenia samochodu. Równie dobrze moglibyśmy napisać w ten sposób logikę, która tworzy tabliczkę czekolady.

Rozwiązanie

Możemy rozumieć ten wzorzec jako łańcuch - `LinkedList`, gdzie elementami są kolejne procesory danego zagadnienia i każdy z nich jak skończy to przekazuje "pałeczkę" dalej. Ilość takich procesorów nie jest w żaden sposób ograniczona. Możemy do tego wprowadzić obiekt wiadomości, która jest przesyłana pomiędzy takimi procesorami. W momencie gdy dany "procesor" zakończy swoje działanie, wysyła on wiadomość do kolejnego "procesora".

Aby wykorzystać ten wzorzec, określimy 3 rodzaje obiektów:

- **Handler** - interface, który odbiera żądanie od **Klienta**, a następnie wysyła takie żądanie do obsługi. Handler wie, tylko o pierwszym ogniwie w łańcuchu i do niego wysyła prośbę o przetworzenie zapytania **Klienta**. Nie ma natomiast pojęcia o kolejnych ogniwach łańcucha. To ogniwa wiedzą o sobie nawzajem (przykład z `LinkedList`),
- **Konkretny Handler** - konkretny Handler, który jest odpowiedzialny za przetworzenie konkretnego zapytania, które może być skierowane albo od **Klienta**, albo od innego Handlera,
- **Klient** - obiekt, który jest źródłem zapytania, to on rozpoczyna całą interakcję w łańcuchu.

Klient przychodzi z pewnym problemem i wysyła go do pierwszego ogniwa w łańcuchu, to ogniwo stara się przeprocesować problem, jeżeli nie jest w stanie, przekazuje problem do następnego ogniwa w łańcuchu. W ten sposób możemy dojść do końca łańcucha i albo przeprocesować problem albo i nie ☺.

Przykład w kodzie

W pierwszym kroku stwórzmy interface `CarHandler`, który będzie implementowany przez każde konkretne ogniwo w łańcuchu. Interface taki powinien określać metodę która pozwala na zdefiniowanie następnego ogniwa w łańcuchu - `setNextHandler()` oraz metodę, która definiuje faktyczną logikę w danym handlerze - `handle()`.

Interface CarHandler

```
public interface CarHandler {

    void setNextHandler(CarHandler nextHandler);
```

```
    void handle(Car car);
}
```

Wiadomością, która będzie przekazywana przez wszystkie ogniwa łańcucha będzie obiekt `Car`. W obiekcie tym zawrzemy wszystkie parametry, które mają być uwzględnione przy produkcji takiego samochodu.

Klasa Car

```
@Data
@With
@AllArgsConstructor(staticName = "of")
public class Car {

    private Type type;

    private String color;

    private List<String> equipment;

    enum Type {
        CABRIOLET,
        JEEP
    }
}
```

Na tym etapie powinniśmy mieć przemyślane, jakie konkretnie ogniwa łańcucha powinny zostać określone aby przejść od początku do końca procesu. Na potrzeby tego przykładu wyróżnimy ogniwa takie jak stworzenie szkieletu samochodu typu `Cabriolet`, stworzenie szkieletu samochodu typu `Jeep`, pomalowanie szkieletu samochodu, dodanie wyposażenia oraz kontrolę jakości. Dlatego też zdefiniujemy handlery takie jak poniżej.

Klasa CabrioletHandler

```
public class CabrioletHandler implements CarHandler {

    private CarHandler carHandler;

    @Override
    public void setNextHandler(final CarHandler nextHandler) {
        this.carHandler = nextHandler;
    }

    @Override
    public void handle(final Car car) {
        if (Car.Type.CABRIOLET.equals(car.getType())) {
            System.out.println("Preparing Cabriolet Skeleton");
            final List<String> cabrioletEquipment = new ArrayList<>(car.getEquipment());
            cabrioletEquipment.add("Cabriolet roof");
            this.carHandler.handle(car.withEquipment(cabrioletEquipment));
        } else {
            this.carHandler.handle(car);
        }
    }
```

```
}
```

Klasa JeepHandler

```
public class JeepHandler implements CarHandler {

    private CarHandler carHandler;

    @Override
    public void setNextHandler(final CarHandler nextHandler) {
        this.carHandler = nextHandler;
    }

    @Override
    public void handle(final Car car) {
        if (Car.Type.JEEP.equals(car.getType())) {
            System.out.println("Preparing Jeep Skeleton");
            final List<String> jeepEquipment = new ArrayList<>(car.getEquipment());
            jeepEquipment.add("Spare Wheel");
            this.carHandler.handle(car.withEquipment(jeepEquipment));
        } else {
            this.carHandler.handle(car);
        }
    }

}
```

Klasa ColorHandler

```
public class ColorHandler implements CarHandler {

    private CarHandler carHandler;

    @Override
    public void setNextHandler(final CarHandler nextHandler) {
        this.carHandler = nextHandler;
    }

    @Override
    public void handle(final Car car) {
        if (Objects.nonNull(car.getColor())) {
            System.out.printf("Preparing %s color%n", car.getColor());
        }
        this.carHandler.handle(car);
    }

}
```

Klasa EquipmentHandler

```
public class EquipmentHandler implements CarHandler {

    private CarHandler carHandler;

    @Override
    public void setNextHandler(final CarHandler nextHandler) {
        this.carHandler = nextHandler;
```

```
    }

    @Override
    public void handle(final Car car) {
        if (Objects.nonNull(car.getEquipment())) {
            System.out.printf("Preparing equipment: %s%n", car.getEquipment());
        }
        this.carHandler.handle(car);
    }

}
```

Klasa QualityCheckHandler

```
public class QualityCheckHandler implements CarHandler {

    private CarHandler carHandler;

    @Override
    public void setNextHandler(final CarHandler nextHandler) {
        this.carHandler = nextHandler;
    }

    @Override
    public void handle(final Car car) {
        System.out.println("Checking quality!\n");
    }
}
```

Każdy w wymienionych wyżej handlerów definiuje logikę, w której próbuje wykonać operacje do której został zaprojektowany - malowanie samochodu, dodanie wyposażenia itp. Jeżeli nie jest w stanie obsłużyć tej logiki (na podstawie jakiś warunków) przekazuje wiadomość (obiekt klasy `Car`) dalej, do następnego procesora. Oczywiście może w międzyczasie taką wiadomość zmodyfikować. Można inaczej to zrozumieć w ten sposób, że każdy procesor (**handler**) próbuje przetworzyć część całości łańcucha. Jeżeli mu się to udaje to procesuje to co jest w stanie i przekazuje wykonanie programu dalej, do innego procesora. Jeżeli nie jest w stanie wykonać takiego przetworzenia, bo nie są spełnione jakieś warunki, dany procesor przekazuje wykonanie programu dalej do innego procesora bez wykonania swojej logiki.

Został nam w tym momencie ostatni krok - powiązanie handlerów ze sobą. Zwróć uwagę, że do tego momentu definiowaliśmy handlery, ale nie określiliśmy w jakiej kolejności mają być one wywołane. Zrobimy to poniżej.

Klasa ChainRunner

```
public class ChainRunner {

    private final CarHandler handler;

    public ChainRunner() { ①
        this.handler = new CabrioletHandler();
        CarHandler handler2 = new JeepHandler();
        CarHandler handler3 = new ColorHandler();
        CarHandler handler4 = new EquipmentHandler();
        CarHandler handler5 = new QualityCheckHandler();

        handler.setNextHandler(handler2);
```

```
        handler2.setNextHandler(handler3);
        handler3.setNextHandler(handler4);
        handler4.setNextHandler(handler5);
    }

    public static void main(String[] args) {
        ChainRunner chainRunner = new ChainRunner();
        chainRunner.handler.handle(Car.of(Car.Type.CABRIOLET, "Blue", List.of("Steering wheel")));
        chainRunner.handler.handle(Car.of(Car.Type.JEEP, "Black", List.of("Wheels")));
    }
}
```

① Jak widzisz, konfiguracja została zdefiniowana w konstruktorze. A co z DIP? Takie przykłady (naginające pokazane zasady) również będą się zdarzać i możesz je traktować jako potencjalne easter eggi, żeby ćwiczyć Twoją spostrzegawczość. Jest to też nawiązanie do praktyki, tzn. wszystkie wymienione zasady są ważne i pomagają w życiu programisty (ktoś je przecież po coś wymyślił). Spotkasz jednak na swojej ścieżce zawodowej wiele miejsc i przykładów, które można by było napisać i przygotować inaczej, bo są niezgodne z niektórymi zasadami. W tym przypadku moglibyśmy przykładowo wydzielić ten kod do osobnej klasy konfiguracyjnej.

Dopiero w klasie `ChainRunner` zdefiniowaliśmy połączenie pomiędzy konkretnymi handlerami.

Możesz zwrócić uwagę, że zarówno **Template method** jak i **Chain of Responsibility** odnoszą się do określenia kroków wykonania jakiegoś procesu. W tym rozumieniu są one do siebie podobne. Można natomiast wyobrazić sobie, że **Chain of Responsibility** jest bardziej rozproszone, co może przekładać się na jego większą elastyczność.

Zalety tego podejścia:

- Klient nie ma pojęcia jakie procesory i w jakiej kolejności będą wykonywały zdefiniowaną logikę. Inaczej mówiąc, jesteśmy tutaj elastyczni bo jakakolwiek zmiana w procesorach jest przezroczysta z perspektywy klienta, on tylko wysyła wiadomość i chce mieć ją przeprocesowaną.
- Możemy podzielić skomplikowaną logikę na kroki co ułatwi zrozumienie konkretnych części procesu. Nie umieszczamy w takiej sytuacji wszystkiego w jednej klasie - wzrasta czytelność kodu.
- Każdy obiekt jest odpowiedzialny za swoją część procesu - **Single Responsibility Principle**.
- W łatwy sposób możemy dodawać kroki do naszej logiki (nawet w środku łańcucha), wystarczy, że stworzymy nowy handler i dowiążemy go do istniejącego łańcucha. Dla klienta jest to niewidoczne, tak samo jak dla innych handlerów, nie musimy ich modyfikować aby wprowadzić zmiany w całej logice.

Na co zwrócić szczególną uwagę?

- Aby procesory nie wpadły w **recursive dependency** (zależność rekurencyjną) z której nie będą w stanie się wydostać. Czyli musimy uważać na sytuację np. handler1 > handler2 > handler1 > handler2 > ...
- Uwzględnienie wszystkich zdefiniowanych procesorów. Musimy zwrócić uwagę, żeby nie pominąć żadnego z nich, albo nie duplikować niektórych definicji.
- Musimy uważać na `NullPointerException`, które może zostać wyrzucone, gdy nie zdefiniujemy następnego handlera.
- Dzięki stosowaniu tego wzorca możemy osiągnąć **luźne powiązanie** (*loose coupling*) pomiędzy

klasami. Możemy natomiast przesadzić w drugą stronę i stworzyć koszmar w łączeniu tych handlerów i zarządzaniu nimi.

Generyczna implementacja

Pokazany przykład skupia się wokół klasy `Car`, która reprezentuje wiadomość, która jest przesyłana pomiędzy handlerami. Możemy natomiast zaimplementować taki **handler** w sposób generyczny.

Klasa GenericHandler

```
public abstract class GenericHandler<T> {

    protected GenericHandler<T> handler;

    public void setNextHandler(GenericHandler<T> nextHandler) {
        this.handler = nextHandler;
    }

    public T handle(T input) {
        T r = handleInput(input);
        if (handler != null) {
            return handler.handle(r);
        }
        return r;
    }

    abstract protected T handleInput(T input);
}
```

Następnie dołożyć do tego implementacje klasy `GenericHandler`.

Klasa TextBugHandler

```
public class TextBugHandler extends GenericHandler<String>{
    @Override
    protected String handleInput(final String input) {
        return input.replace("w zadaniu jest błąd", "w zadaniu nie ma błędu");
    }
}
```

Klasa TextOpinionHandler

```
public class TextOpinionHandler extends GenericHandler<String>{
    @Override
    protected String handleInput(final String input) {
        return input.replace("mam złą opinię o bootcampie", "bootcamp jest super");
    }
}
```

Klasa TextZajavkaHandler

```
public class TextZajavkaHandler extends GenericHandler<String> {

    @Override
```

```
    protected String handleInput(final String input) {
        return input.replace("zajavka nie jest super", "zajavka jest super!");
    }
}
```

Jeżeli teraz uruchomimy ten program w sposób pokazany poniżej, to zaimplementowaliśmy wzorzec **Chain of Responsibility** w sposób generyczny.

Klasa GenericRunner

```
public class GenericRunner {

    public static void main(String[] args) {
        GenericHandler<String> handler1 = new TextZajavkaHandler();
        GenericHandler<String> handler2 = new TextBugHandler();
        GenericHandler<String> handler3 = new TextOpinionHandler();
        handler1.setNextHandler(handler2);
        handler2.setNextHandler(handler3);

        String result = handler1.handle(
            "Wziąłem udział w bootcampie i powiem wam, że zajavka nie jest super. " +
            "Zrobiłem zadanie 2 i w zadaniu jest błąd. " +
            "Podsumowując, mam złą opinię o bootcampie!");
        System.out.println(result);
    }
}
```

Sposób działania jest analogiczny do opisanego w przykładzie z produkcją samochodów.

Dokładamy Java 8

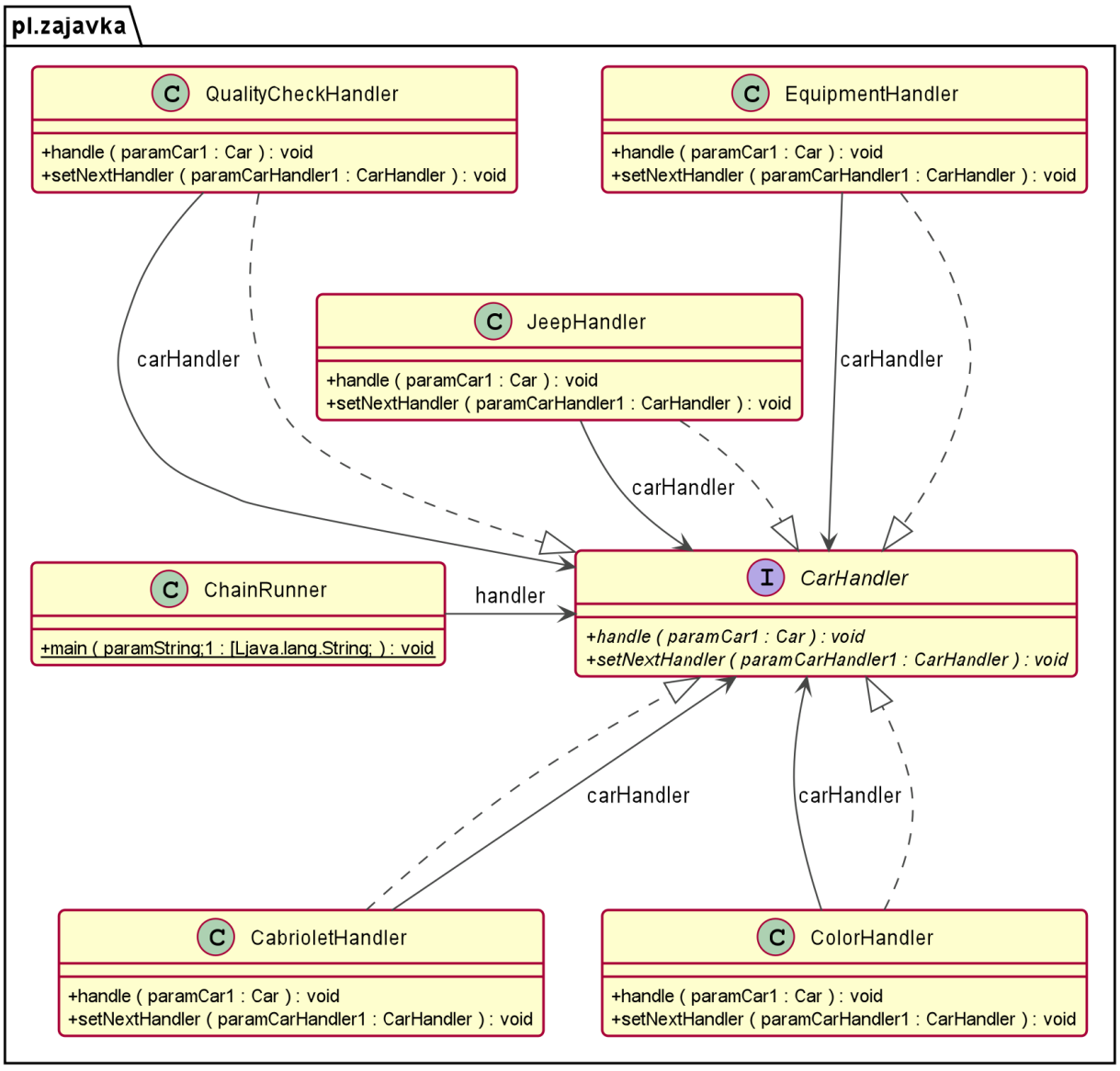
Jeżeli umiemy już zaimplementować wzorzec **Chain of Responsibility** korzystając z generycznego handlera - możemy spróbować napisać to samo przy wykorzystaniu lambd. Każdy z handlerów możemy zaimplementować w postaci lambdy (zwróć uwagę, że każdy z handlerów implementuje tylko jedną metodę abstrakcyjną). Następnie możemy wywołanie każdego z handlerów złączyć przy wykorzystaniu interfejsu `Function`. W tym celu wykorzystamy metodę `andThen()`, która pozwala na złączenie ze sobą wywołań kolejnych implementacji interfejsu `Function`. Interface `UnaryOperator` dziedziczy z interfejsu `Function` stąd możemy go tutaj użyć. Przykład poniżej.

```
public class LambdaChain {

    public static void main(String[] args) {

        UnaryOperator<String> textZajavkaHandler = (String input) ->
            input.replace("zajavka nie jest super", "zajavka jest super!");
        UnaryOperator<String> textBugHandler = (String input) ->
            input.replace("w zadaniu jest błąd", "w zadaniu nie ma błędu");
        UnaryOperator<String> textOpinionHandler = (String input) ->
            input.replace("mam złą opinię o bootcampie", "bootcamp jest super");

        Function<String, String> pipeline = textZajavkaHandler
            .andThen(textBugHandler)
            .andThen(textOpinionHandler);
```

```
        String input = "Wziąłem udział w bootcampie i powiem wam, że zajavka nie jest super. " +
            "Zrobiłem zadanie 2 i w zadaniu jest błąd. " +
            "Podsumowując, mam złą opinię o bootcampie!";
        String result = pipeline.apply(input);
        System.out.println(result);
    }
}
```

Diagram UML

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



Obraz 2. UML Chain of Responsibility Class Diagram

Observer

Problem jaki rozwiązuje

Nazwa tego wzorca (**Obserwator**) jest związana z obserwacją zmian stanu obiektu. Możemy wykorzystać ten wzorzec jeżeli zależy nam na obserwowaniu zmiany stanu jakiegoś obiektu i odpowiedniej reakcji jeżeli taki stan ulegnie zmianie. Wyróżniamy tutaj obiekt **Obserwatora**, który jest odpowiedzialny za obserwowanie zmiany stanu innego obiektu, który jest nazywany **Subject** (czasami jest również określany jako **Observable**).

Przykład z życia

Przykładem z życia jest newsletter, na który możemy się zapisać jako obserwator. Możemy zarejestrować się w danym newsletterze i dostawać powiadomienia w momencie gdy np. na jakiejś stronie internetowej pojawi się nowa treść - zmianie ulega wtedy stan zawartości takiej strony. Możemy się z takiego newslettera wyrejestrować.

Rozwiązanie

Aby zastosować ten wzorzec należy stworzyć klasę np. `Observer`, która będzie odpowiedzialna za obserwację zmian stanu innego obiektu - **Subject**. **Subject** będzie zawierał listę swoich obserwatorów. Jeżeli stan obiektu **Subject** ulegnie zmianie - zostaną automatycznie powiadomieni wszyscy obserwatorzy dodani do tej listy. Obserwatorzy mogą się zarejestrować i wyrejestrować z listy obserwatorów. **Subject** zawiera również metodę, dzięki której możemy dokonać powiadomienia wszystkich obserwatorów. **Obserwator** powinien natomiast zawierać metodę, która ma zostać wywołana w momencie zmiany stanu obiektu **Subject**.

Przykład w kodzie

Przykład w kodzie będzie opierał się o serwis internetowy, który powiadamia swoich obserwatorów, którzy zarejestrowali się na listę powiadomień. Zacznijmy zatem od zdefiniowania interfejsu `Subject`.

Interface Subject

```
public interface Subject {

    void register(Observer observer); ①

    void unregister(Observer observer); ②

}
```

W interface `Subject` określamy dwie metody służące do zarejestrowania i wyrejestrowania z listy powiadomień, oznaczone jako 1 i 2.

Następnie dodajemy implementację tego interfejsu, w tym przypadku klasę `NewsTopic`. Czyli będzie to klasa reprezentująca jakiś temat, którym są zainteresowani obserwatorzy. W momencie, gdy w temacie pojawią się nowe wiadomości - zostają powiadomieni obserwatorzy.

Klasa NewsTopic

```
public class NewsTopic implements Subject {

    private final List<Observer> observers = new ArrayList<>();

    private String news;

    @Override
    public void register(Observer observer) {
        if (Objects.isNull(observer)) {
            throw new NullPointerException("Observer is null");
        }
        if (!observers.contains(observer)) {
            observers.add(observer);
        }
    }

    @Override
    public void unregister(Observer observer) {
        observers.remove(observer);
    }

    public void setNews(String message) {
        System.out.printf("Message posted: %s%n", message);
        this.news = message;
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

Klasa `NewsTopic` pozwala na zarejestrowanie i wyrejestrowanie obserwatora z listy powiadomień. Gdy zmieni się stan klasy `NewsTopic`, czyli pojawi się nowy news (wywołana zostanie metoda `setNews()`) automatycznie powiadomieni zostaną wszyscy obserwatorzy, którzy zarejestrowali się przy wykorzystaniu metody `register()`.

Do tego zdefiniujmy interface `Observer`.

Interface Observer

```
public interface Observer {

    void update(final String message);
}
```

Oraz dodajmy klasę `NewsSubscriber`, która implementuje interface `Observer`. Obiekty klasy `NewsSubscriber` zostaną powiadomione gdy nastąpi wywołanie metody `setNews()` w klasie `NewsTopic`.

Klasa NewsSubscriber

```
public class NewsSubscriber implements Observer {

    private final String name;

    public NewsSubscriber(String name) {
```

```
        this.name = name;
    }

    @Override
    public void update(final String message) {
        if (message == null) {
            System.out.printf("%s - Message null%n", name);
        } else
            System.out.printf("%s - Message received - %s%n", name, message);
    }

}
```

Wywołajmy teraz ten kod.

Klasa NewsRunner

```
public class NewsRunner {

    public static void main(String[] args) {
        NewsTopic subject = new NewsTopic();

        Observer observer1 = new NewsSubscriber("NewsSubscriber1");
        Observer observer2 = new NewsSubscriber("NewsSubscriber2");
        Observer observer3 = new NewsSubscriber("NewsSubscriber3");

        subject.register(observer1);
        subject.register(observer2);
        subject.register(observer3);

        subject.setNews("News Arrived!");
    }
}
```

Gdy uruchomisz teraz ten kod to zaobserwujesz, że gdy obiekt `NewsTopic` otrzyma wiadomość "News Arrived!", to każdy zarejestrowany obserwator zostanie o tym powiadomiony i odbierze tę wiadomość.

Zalety tego podejścia:

- Możemy wykorzystać ten wzorzec aby rejestrować dowolną liczbę obserwatorów.
- Wzorzec ten może być wykorzystany gdy chcemy powiadomić pewną ilość obserwatorów o zaistnieniu pewnego wydarzenia.

Dokładamy Java 8

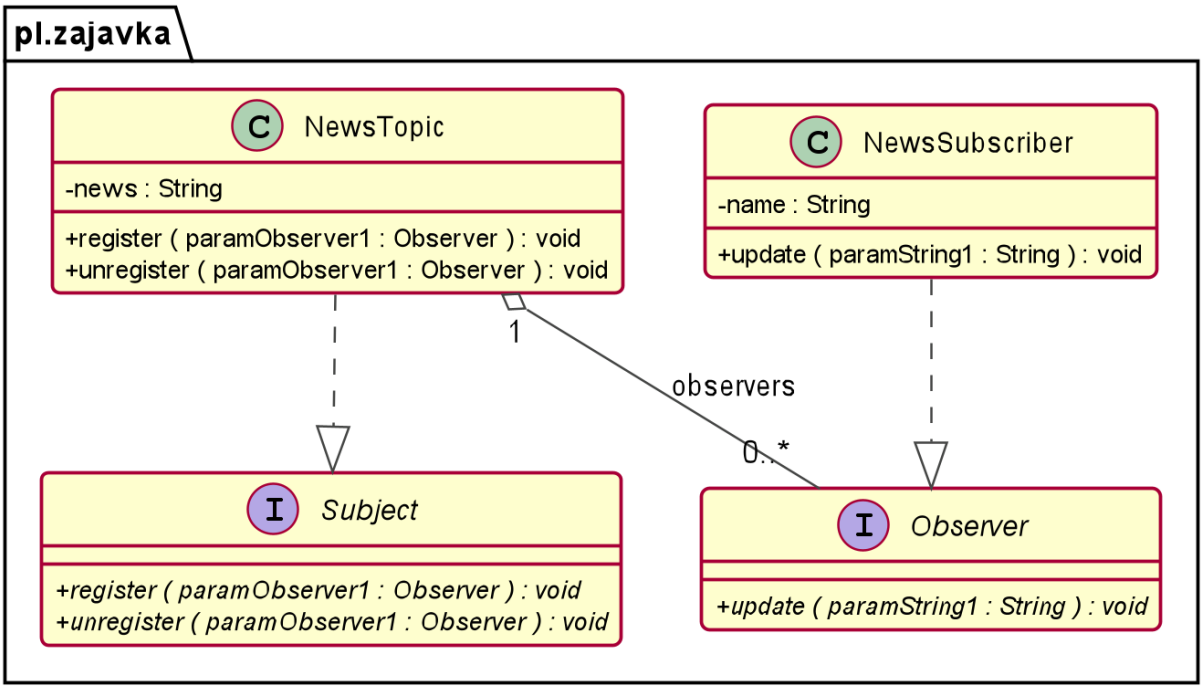
Zwróć uwagę, że interface `Observer` jest funkcyjny, co oznacza, że można go zaimplementować przy wykorzystaniu lambdy.

```
public class NewsRunner {

    public static void main(String[] args) {
        NewsTopic subject = new NewsTopic();
        Observer observer1 = news -> System.out.printf("Message received: %s%n", news);
        subject.register(observer1);
```

```
        subject.setNews("News Arrived!");
    }
}
```

Diagram UML

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



Obraz 3. UML Observer Class Diagram

Strategy

Problem jaki rozwiązuje

Strategia rozwiązuje problem wyboru algorytmu, który ma być wykorzystany w programie przez klienta. Czyli, Ty jako klient danej metody chcesz zdecydować jaki algorytm ma zostać wykorzystany w jej środku. Skoro opisujemy to w ten sposób, to możesz zauważyć, że sytuacja taka będzie potrzebna jeżeli takich algorytmów będziemy mieli dostępnych kilka. Czasami można znaleźć w internecie inną nazwę tego wzorca - **Policy Pattern**.

Przykład z życia

Na tym etapie powinniśmy już kojarzyć metodę `Collections.sort()`. Gdyby spojrzeć na nią jak na wzorzec **Strategii** to zauważymy, że argumentem wywołania tej metody jest konkretny algorytm jaki ma stać za porównaniem elementów kolekcji. To my wywołując tę metodę określamy sposób sortowania.

Przykładem z życia natomiast może być sklep internetowy, w którym to klient wybiera sposób dostawy towaru - Kurier, Poczta lub Paczkomat.

Rozwiązanie

Stworzymy interface `DeliveryStrategy`, który będzie nam określał sposób dostawy paczki. Następnie na tej podstawie możemy stworzyć konkretne implementacje tego interfejsu, które będą implementowały konkretne sposoby dostawy paczek. Przy wyborze dostawy, argument metody, która realizuje dostawę będzie określony przez interfejs, natomiast to konkretna implementacja zapewni nam (jako klientowi metody) możliwość wyboru sposobu dostawy.

Przykład w kodzie

Na początek zdefiniujmy interfejs `DeliveryStrategy`:

Interface DeliveryStrategy

```
public interface DeliveryStrategy {

    void deliver(final Parcel parcel);
}
```

Do tego stwórzmy klasę `Parcel`, która będzie reprezentowała paczkę. Dodamy w tej klasie pole `name` aby nazwać co w danej paczce się znajduje, np. skarpetki, krzesła itp.

Klasa Parcel

```
@Data
public class Parcel {

    private final String name;
}
```

Następnie zaimplementujmy interface `DeliveryStrategy` przy wykorzystaniu klas `CourierStrategy`, `ParcelLockerStrategy` oraz `PostStrategy`.

Klasa CourierStrategy

```
public class CourierStrategy implements DeliveryStrategy {

    @Override
    public void deliver(final Parcel parcel) {
        System.out.printf("Parcel %s delivered by Courier%n", parcel);
    }
}
```

Klasa ParcelLockerStrategy

```
public class ParcelLockerStrategy implements DeliveryStrategy {

    @Override
    public void deliver(final Parcel parcel) {
        System.out.printf("Parcel %s delivered by ParcelLocker%n", parcel);
    }
```

```
}
```

Klasa PostStrategy

```
public class PostStrategy implements DeliveryStrategy {

    @Override
    public void deliver(final Parcel parcel) {
        System.out.printf("Parcel %s delivered by Post%n", parcel);
    }
}
```

Jeżeli mamy już stworzone te klasy - napiszmy klasę `OnlineShop`, która pozwoli nam dodawać paczki do dostarczenia oraz zdefiniuje metodę `deliver()`, która będzie wywoływała algorytm dostarczania paczek.

Klasa OnlineShop

```
public class OnlineShop {

    private List<Parcel> parcels = new ArrayList<>();

    public void addParcel(Parcel parcel) {
        parcels.add(parcel);
    }

    public void deliver(DeliveryStrategy deliveryStrategy) {
        parcels.forEach(deliveryStrategy::deliver);
    }

}
```

Ostatnim krokiem jest stworzenie klasy `OnlineShopRunner`, która pozwoli nam uruchomić program i przetestować **Strategy Pattern**.

Klasa OnlineShopRunner

```
public class OnlineShopRunner {

    public static void main(String[] args) {
        OnlineShop onlineShop = new OnlineShop();

        onlineShop.addParcel(new Parcel("skarpetki"));
        onlineShop.addParcel(new Parcel("monitory"));
        onlineShop.addParcel(new Parcel("samochody"));

        System.out.println("\n##CourierStrategy##");
        onlineShop.deliver(new CourierStrategy());

        System.out.println("\n##PostStrategy##");
        onlineShop.deliver(new PostStrategy());

        System.out.println("\n##ParcelLockerStrategy##");
        onlineShop.deliver(new ParcelLockerStrategy());
    }
}
```

W klasie `OnlineShopRunner` początkowo tworzymy instancję sklepu internetowego oraz dodajemy do niego paczki, które zakupiliśmy w tym sklepie. Paczki mogą w sobie zawierać różne przedmioty - krzesła, samochody, skarpetki, dlatego określiliśmy pole `name` w klasie `Parcel`. Dodajemy kolejne paczki do naszego sklepu internetowego. Gdy wszystkie paczki są już dodane możemy spróbować je dostarczyć. Metoda `deliver()`, w klasie `OnlineShop` przyjmuje interface `DeliveryStrategy`. Dzięki zastosowaniu takiego podejścia możemy przy wywołaniu tej metody zapewnić konkretną implementację interfejsu `DeliveryStrategy`, czyli wybrać strategię dostarczenia paczki. Robimy to jako klient metody `deliver()`, czyli możemy w **Runtime** określić, jaki sposób dostarczania paczek nas interesuje i wybrać ten sposób określając konkretną implementację interfejsu `DeliveryStrategy`.

Zalety tego podejścia:

- Wzorzec **Strategii** jest bardzo wygodny gdy chcemy wyjąć "fragment" logiki poza metodę i na zewnątrz tej metody zdecydować o algorytmie, który ma zostać uruchomiony w środku.
- Wzorzec ten daje nam dużą elastyczność gdy mamy kilka algorytmów do wyboru i w trakcie działania programu chcemy wybrać jeden z nich.

Dokładamy Java 8

Gdy pojawiła się Java 8 i lambdy, to również i ten wzorzec został uproszczony. Zamiast definiować klasy takie jak `CourierStrategy`, `ParcelLockerStrategy` czy `PostStrategy` możemy zastąpić implementacje tych klas lambdą. Klasa `OnlineShop` mogłaby wtedy wyglądać tak jak w przykładzie poniżej. Oczywiście nie musimy zawsze wykorzystywać interfejsu funkcyjnego `Consumer`. Możemy tutaj wykorzystać różne interfejsy funkcyjne w zależności od naszych potrzeb.

```
public class OnlineShop {

    private List<Parcel> parcels = new ArrayList<>();

    public void addParcel(Parcel parcel) {
        parcels.add(parcel);
    }

    public void deliver(Consumer<Parcel> deliveryStrategy) {
        parcels.forEach(deliveryStrategy);
    }

}
```

Nie musimy wtedy tworzyć klas `CourierStrategy`, `ParcelLockerStrategy` czy `PostStrategy`. Możemy zaimplementować je przy wykorzystaniu lambdy.

Klasa OnlineShopRunner

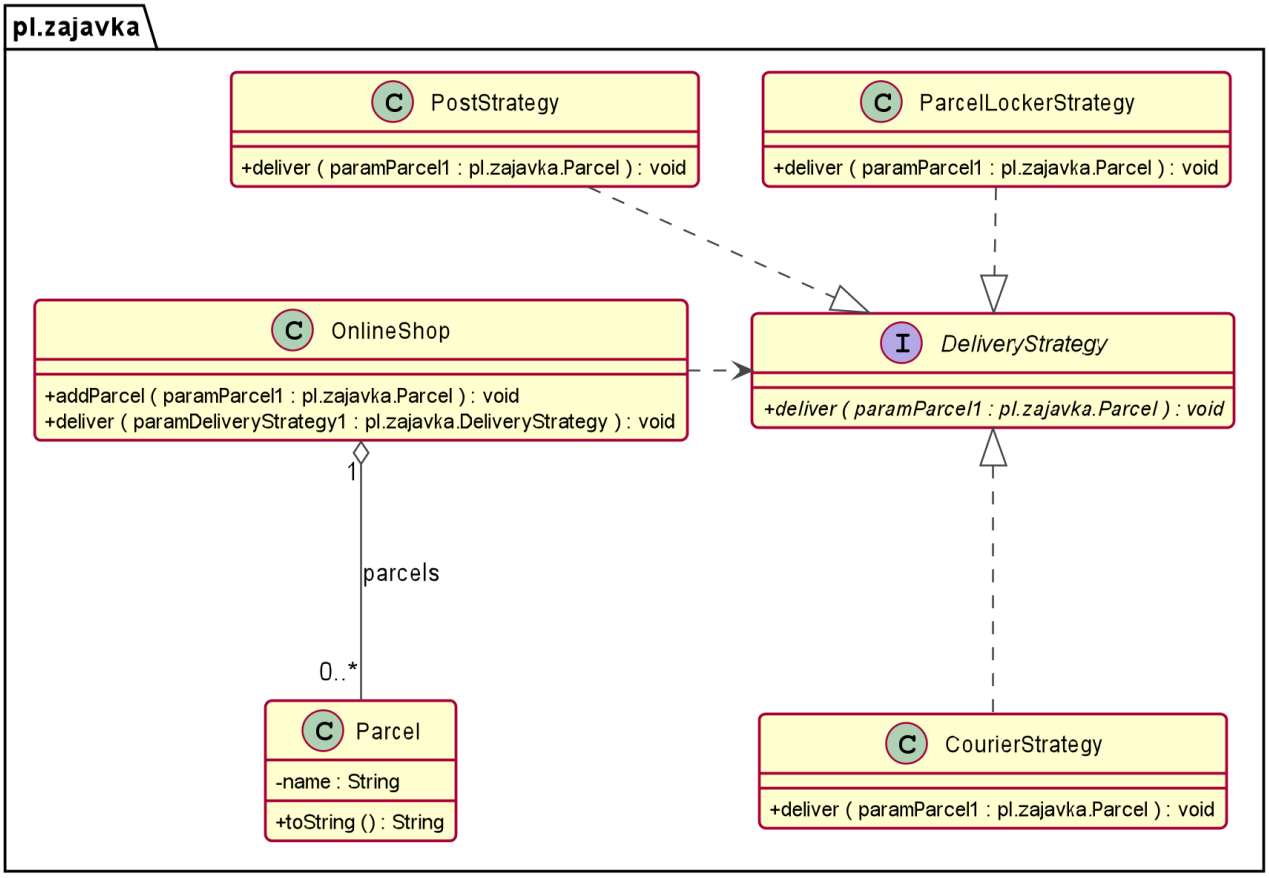
```
public class OnlineShopRunner {

    public static void main(String[] args) {
        OnlineShop onlineShop = new OnlineShop();

        onlineShop.addParcel(new Parcel("skarpetki"));
        onlineShop.addParcel(new Parcel("monitory"));
        onlineShop.addParcel(new Parcel("samochody"));
```

```
        System.out.println("\n##CourierStrategy##");
        onlineShop.deliver(parcel -> System.out.printf("Parcel %s delivered by Courier%n", parcel));

        System.out.println("\n##PostStrategy##");
        onlineShop.deliver(parcel -> System.out.printf("Parcel %s delivered by Post%n", parcel));

        System.out.println("\n##ParcelLockerStrategy##");
        onlineShop.deliver(parcel -> System.out.printf("Parcel %s delivered by ParcelLocker%n", parcel));
    }
}
```

Jak widać na załączonym przykładzie - podejście funkcyjne i stosowanie lambd upraszcza i skraca zapisany kod.

Diagram UML

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



Obraz 4. UML Strategy Class Diagram

Visitor

Problem jaki rozwiązuje

Visitor jest wzorcem, który jest stosowany w przypadku, gdy mamy wywołać jakąś operację na grupie podobnych do siebie obiektów. Zamiast definiować bardzo podobne zachowanie dla każdego ze wspomnianych obiektów, możemy przenieść podobną grupę zachowań w jedno miejsce. Zwiększa to czytelność, bo podobna grupa zachowań jest wtedy dostępna w jednym miejscu w kodzie, a nie w wielu klasach. Innymi słowy, wyciągamy logikę, która nas interesuje, z obiektu na zewnątrz.

W ten sposób stosujemy się do zasady **Open/Close**, gdyż nie będziemy musieli modyfikować kodu obiektu, który nas interesuje. Możemy natomiast zastosować wtedy nowego **Visitora**, dzięki czemu rozszerzamy implementację, a nie ją modyfikujemy.

Przykład z życia

Często poruszanym przykładem jest dodawanie produktów w sklepie do koszyka. Dodając produkty do koszyka wykonujemy bardzo podobną operację dla każdego z produktów - dodajemy go do koszyka 😀. Gdy dojdziemy już do kasy należy zliczyć końcową kwotę jaką zapłacimy za produkty. I tutaj pojawia się możliwość dla **Visitora**, albo możemy mieć logikę liczącą końcową cenę produktu w każdym z tych obiektów oddzielnie, albo możemy ją przenieść w jedno miejsce, w którym będziemy zliczali wartość końcową każdego z zakupionych dóbr. Końcowa kwota każdego z produktów może być nieco bardziej skomplikowana jeżeli np. będziemy chcieli naliczyć zniżkę na rower, albo produkt jest sprzedawany w kilogramach i trzeba przeliczyć cenę końcową w zależności od wagi.

Rozwiązanie

Rozwiązaniem jest stworzenie interfejsu np. `ShoppingCartElement`, który będzie definiował metodę `accept()`, która przyjmie klasę `Visitor`, np. `ShoppingCartVisitor`. Klasa **Visitor** będzie odpowiedzialna za zliczanie końcowej kwoty każdego z produktów.

Przykład w kodzie

W pierwszej kolejności stwórzmy interface `ShoppingCartElement`, który będzie reprezentował wszystkie elementy jakie możemy dodać do koszyka w naszym sklepie.

Interface ShoppingCartElement

```
public interface ShoppingCartElement {
    BigDecimal accept(ShoppingCartVisitor visitor);
}
```

Dodajmy teraz elementy, które będą implementowały ten interface, np. `Apple` oraz `Bicycle`.

Klasa Apple

```
@Data
@AllArgsConstructor
public class Apple implements ShoppingCartElement {

    private BigDecimal pricePerKg;
```

```
    private BigDecimal weight;

    @Override
    public BigDecimal accept(final ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Klasa Bicycle

```
@Data
@AllArgsConstructor
public class Bicycle implements ShoppingCartElement {

    private BigDecimal price;

    private BigDecimal discount;

    @Override
    public BigDecimal accept(final ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Zwróć uwagę, na to, że metoda `accept()` wywołuje metodę `visit()` z interfejsu **Visitor** i przekazuje do niej samą siebie. Możemy teraz stworzyć interface `ShoppingCartVisitor`, który powinien posiadać metody `visit()` odpowiadające tym wywołanym w klasach `Bicycle` i `Apple`.

```
public interface ShoppingCartVisitor {

    BigDecimal visit(Apple apple);

    BigDecimal visit(Bicycle bicycle);
}
```

Możemy teraz zaimplementować interface `ShoppingCartVisitor` i określić w tej implementacji logikę naliczania ceny końcowej dla wspomnianych produktów `Apple` i `Bicycle`.

Klasa ShoppingCartVisitorImpl

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

    @Override
    public BigDecimal visit(final Apple apple) {
        BigDecimal totalCost = apple.getWeight().multiply(apple.getPricePerKg());
        System.out.println("Calculated Apple cost: " + totalCost);
        return totalCost;
    }

    @Override
    public BigDecimal visit(final Bicycle bicycle) {
        BigDecimal totalCost = bicycle.getPrice().subtract(bicycle.getDiscount());
        System.out.println("Calculated Bicycle cost: " + totalCost);
        return totalCost;
```

```
    }
}
```

Możemy teraz wywołać napisaną logikę z klasy `VisitorRunner`.

```
public class VisitorRunner {

    public static void main(String[] args) {
        List<ShoppingCartElement> elements = List.of(
            new Bicycle(BigDecimal.valueOf(100.25), BigDecimal.TEN),
            new Bicycle(BigDecimal.valueOf(420.99), BigDecimal.valueOf(10.20)),
            new Apple(BigDecimal.valueOf(9.80), BigDecimal.valueOf(2.34)),
            new Apple(BigDecimal.valueOf(9.80), BigDecimal.valueOf(6.12))
        );

        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();

        BigDecimal totalCost = elements.stream()
            .map(element -> element.accept(visitor))
            .reduce(BigDecimal.ZERO, BigDecimal::add);

        System.out.println("Total shopping cart cost: " + totalCost);
    }
}
```

Zalety tego podejścia:

- Jeżeli zmianie ulegnie logika, która jest określona w klasie **Visitor**, musimy dokonać zmiany tylko w jednym miejscu w kodzie, a nie w wielu.
- Możemy modyfikować wybraną logikę (tą która jest w **Visitorze**) bez modyfikacji samego obiektu.
- Możemy zaimplementować kilku **Visitorów**, gdzie każdy może implementować logikę na inny sposób, np. jeden wizytor może zapisywać informacje do plików na dysku, a drugi w bazie danych.

Na co zwrócić szczególną uwagę?

- Należy jednak zwracać uwagę na to, że wszystkie metody klasy **Visitor** powinny zwracać ten sam typ (np. interface). W momencie gdy zaczniemy tutaj coś mieszać wprowadzi nam to chaos w kodzie. Często typem zwracanym tych metod jest po prostu `void`.
- Wszędzie duplikujemy implementację metody `accept()`.
- Jeżeli przeniesiemy całą logikę ze wszystkich klas do jednego pliku **Visitora**, utrzymanie tego może stać się ciężkie. Dlatego należy stosować to podejście z głową. Możemy dojść do sytuacji, gdzie będziemy mieli wielu **Visitorów**, wtedy dodanie kolejnego obiektu w projekcie, który jest używany w **Visitorach** będzie oznaczało konieczność modyfikacji każdego **Visitora**.

Dokładamy Java 8

Wcześniej zostało wspomniane, że metoda `accept()` powtarza się w każdej implementacji interfejsu `ShoppingCartElement`. Z racji, że Java 8 pozwoliła nam na definiowanie metod domyślnych w interfejsach, spróbujemy to wykorzystać.

Przepiszemy interface `ShoppingCartElement` w sposób pokazany poniżej.

Interface ShoppingCartElement

```
public interface ShoppingCartElement {

    default BigDecimal accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Klasy `Apple` i `Bicycle` będą teraz wyglądały w ten sposób

Klasa Apple

```
@Data
@AllArgsConstructor
public class Apple implements ShoppingCartElement {

    private BigDecimal pricePerKg;

    private BigDecimal weight;
}
```

Klasa Bicycle

```
@Data
@AllArgsConstructor
public class Bicycle implements ShoppingCartElement {

    private BigDecimal price;

    private BigDecimal discount;

}
```

Następnie przepiszemy interface `ShoppingCartVisitor` w następujący sposób.

Interface ShoppingCartVisitor

```
@FunctionalInterface
public interface ShoppingCartVisitor {

    BigDecimal visit(ShoppingCartElement shoppingCartElement);
}
```

Największa rewolucja nastąpi w klasie, która implementuje `ShoppingCartVisitor`, ale jest to pewnego rodzaju przekłamanie, gdyż nowa klasa `ShoppingCartVisitorMap` nie implementuje bezpośrednio interfejsu `ShoppingCartVisitor`, ale korzysta z niego.

Klasa ShoppingCartVisitorMap

```
public class ShoppingCartVisitorMap
    implements Function<Class<? extends ShoppingCartElement>, ShoppingCartVisitor> { ①
```

```
    private static final Map<Class<? extends ShoppingCartElement>, ? extends ShoppingCartVisitor>
        VISITORS = Map.of(
        Apple.class, element -> visit((Apple) element),
        Bicycle.class, element -> visit((Bicycle) element)
    ); ②

    public static BigDecimal visit(final Apple apple) {
        BigDecimal totalCost = apple.getWeight().multiply(apple.getPricePerKg());
        System.out.println("Calculated Apple cost: " + totalCost);
        return totalCost;
    }

    public static BigDecimal visit(final Bicycle bicycle) {
        BigDecimal totalCost = bicycle.getPrice().subtract(bicycle.getDiscount());
        System.out.println("Calculated Bicycle cost: " + totalCost);
        return totalCost;
    }

    @Override
    public ShoppingCartVisitor apply(final Class<? extends ShoppingCartElement> aClass) { ③
        return VISITORS.get(aClass);
    }
}
```

① Rewolucja polega na tym, że w klasie ShoppingCartVisitorMap nie implementujemy teraz interfejsu ShoppingCartVisitor, tylko Function. Funkcja jako parametr wejściowy definiuje klasę, która korzysta z upper-bound wildcard aby określić, że klasy mają implementować ShoppingCartElement. Parametrem wyjściowym z Function, jest implementacja ShoppingCartVisitor, dlatego Function określa ten interface jako typ generyczny.

② W środku klasy ShoppingCartVisitorMap definiujemy mapę, która jako klucz ma konkretną wartość klasy - Apple.class lub Bicycle.class, natomiast jako wartość definiuje implementację interfejsu funkcyjnego ShoppingCartVisitor - bo ShoppingCartVisitor jest interfejsem funkcyjnym. Czyli Mapa wygląda w ten sposób - **klasa:implementacja**.

③ Przez to, że klasa ShoppingCartVisitorMap implementuje interface Function, musimy zdefiniować metodę apply(), która na podstawie mapy VISITORS zwróci konkretną implementację interfejsu funkcyjnego ShoppingCartVisitor.

Zmieni to lekko implementację klasy VisitorRunner.

Klasa VisitorRunner

```
public class VisitorRunner {

    public static void main(String[] args) {
        List<ShoppingCartElement> elements = List.of(
            new Bicycle(BigDecimal.valueOf(100.25), BigDecimal.TEN),
            new Bicycle(BigDecimal.valueOf(420.99), BigDecimal.valueOf(10.20)),
            new Apple(BigDecimal.valueOf(9.80), BigDecimal.valueOf(2.34)),
            new Apple(BigDecimal.valueOf(9.80), BigDecimal.valueOf(6.12))
        );

        ShoppingCartVisitorMap visitorMap = new ShoppingCartVisitorMap(); ①

        BigDecimal totalCost = elements.stream()
            .map(element -> element.accept(visitorMap.apply(element.getClass()))) ②
```

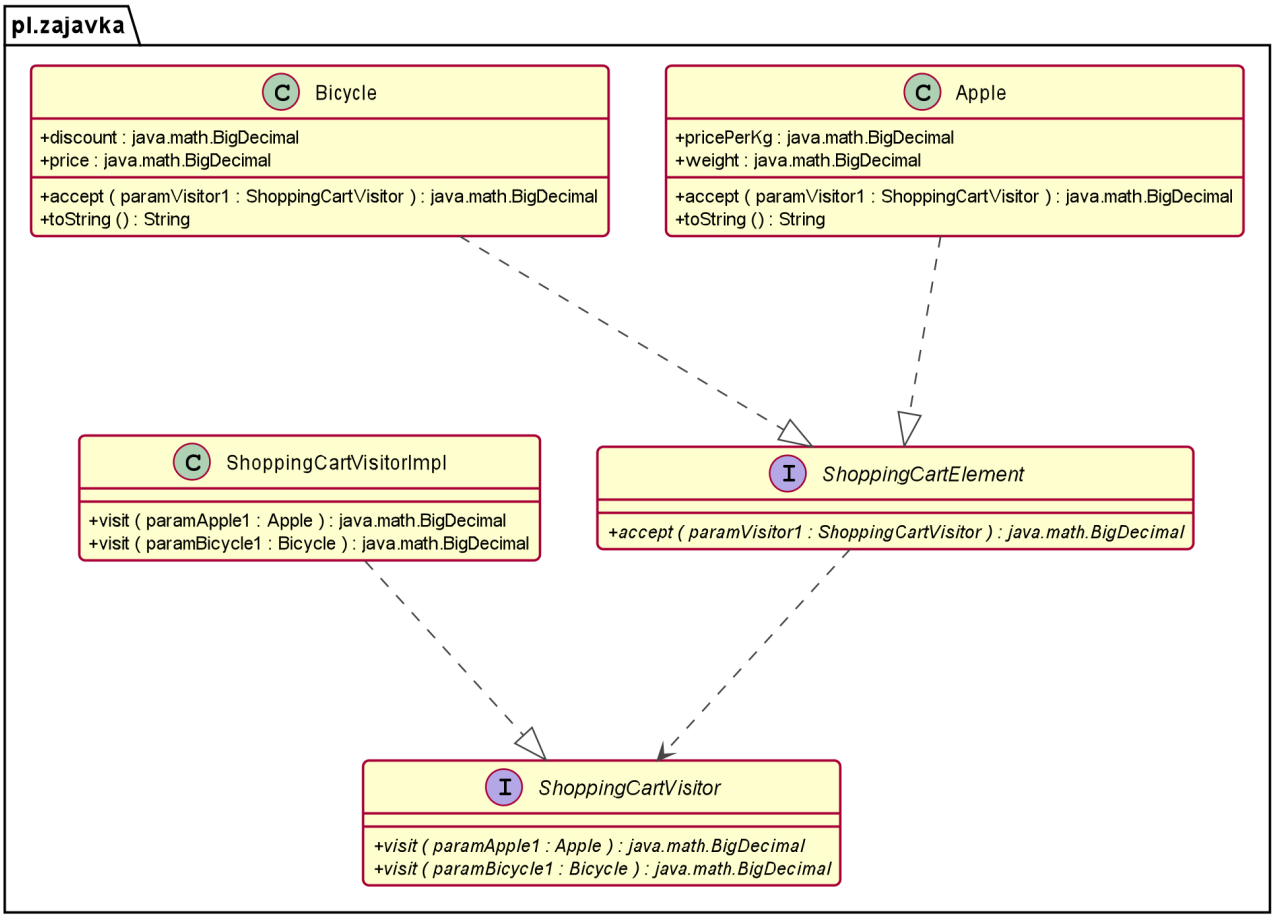
```
            .reduce(BigDecimal.ZERO, BigDecimal::add);

        System.out.println("Total shopping cart cost: " + totalCost);
    }
}
```

① W tym miejscu wykorzystujemy teraz `ShoppingCartVisitorMap` zamiast `ShoppingCartVisitorImpl` tak jak poprzednio.

② W tym miejscu nadal wykorzystujemy metodę `accept()`, natomiast jej argumentem ma być konkretna implementacja `ShoppingCartVisitor`. Konkretne implementacje `ShoppingCartVisitor` są zdefiniowane w mapie `ShoppingCartVisitorMap`. Musimy zatem wewnątrz wywołania `element.accept(X)` pozyskać najpierw `X`. Pozyskanie `X` wymaga wywołania metody `apply()`, która jest zdefiniowana w klasie `ShoppingCartVisitorMap`. W ten sposób dostaniemy konkretną implementację `ShoppingCartVisitor`. Aby dostać tę implementację musimy pobrać element z mapy w `ShoppingCartVisitorMap`. Konkretna implementacja interfejsu funkcyjnego `ShoppingCartVisitor` jest pobierana z mapy na podstawie wartości `Apple.class` lub `Bicycle.class`, dlatego właśnie wywołujemy tutaj `element.getClass()`.

Diagram UML

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



Obraz 5. UML Visitor Class Diagram

Podsumowanie

Wiesz już w tym momencie, że jeżeli ktoś powie **Strategia** albo **Visitor** to oznacza to pewien rodzaj relacji między klasami lub interfejsami. Oczywiście przedstawione wzorce projektowe nie dotyczą tylko Javy, a różnych języków programowania. Każdy język programowania implementuje dany wzorzec projektowy stosując konstrukcje właściwe dla danego języka. Pamiętać należy o tym, że wzorce mówią o powiązaniu ze sobą obiektów, a nie o tym, że ma to być w Javie.

Jeżeli wcześniej pewnego rodzaju konstrukcje, w których łączone były ze sobą klasy, były używane przez Ciebie nieświadomie to wiesz już teraz, że takie relacje mają swoje nazwy. Znajomość wzorców projektowych przydaje się często w komunikacji z innymi ludźmi.

Jak to wszystko spamiętać i nie zwariować? Oczywiście nie należy uczyć się tego kodu na pamięć. W praktyce jeżeli rozumiesz jaki problem dany wzorzec projektowy rozwiązuje to jednocześnie będziesz też w stanie taki kod napisać. Albo będziesz wiedzieć, jakich przykładów kodu należy szukać w Google, żeby rozwiązać Twój problem ☺.