Design Patterns

Spis treści

Structural Design Patterns

Ta grupa wzorców projektowych skupia się wokół różnych form struktury klasy, korzystania z dziedziczenia, kompozycji. Dzięki poniższym wzorcom dowiemy się w jaki sposób możemy powiązać ze sobą klasy w Javie aby rozwiązać pewien określony problem. Przykładowe wzorce **strukturalne** to (wyjaśniamy je po kolei poniżej):

- Adapter
- Composite
- Facade
- Proxy
- Decorator
- Bridge

Przejdźmy teraz do omówienia każdego z nich.

Adapter

Problem jaki rozwiązuje

Jeżeli napotkamy w praktyce sytuację, gdy będziemy mieli 2 interfejsy, które w żaden sposób nie są związane ze sobą (czyli np. będziemy potrzebowali wykorzystać 2 różne biblioteki, ewentualnie 2 moduły z naszego kodu) i będziemy chcieli te interfejsy zachęcić do wspólnej pracy 😄 to możemy wykorzystać ten wzorzec. Inaczej mówiąc, wyobraźmy sobie, że mamy bibliotekę, która dostarcza swoje klasy (swoje API). Jednocześnie przygotowaliśmy wcześniej kod, w którym napisaliśmy jak chcielibyśmy z takiej biblioteki korzystać. Czyli nie znając biblioteki, napisaliśmy swój kod, w którym określiliśmy co chcemy mieć zrobione przez bibliotekę. Dokładnie w takich sytuacjach można zastosować wzorzec zwany **Adapter**.

Sytuacja, która została opisana, (czyli tworzenie swoich modułów kompletnie nie znając jeszcze biblioteki albo systemu, z którym będziemy się integrować) może wystąpić w praktyce jeżeli czekamy na zrobienie jakiegoś modułu systemu przez kogoś "z zewnątrz", ale nie chcemy w tym czasie patrzeć w ścianę. Czyli chcemy się przygotować na kod "z zewnątrz" opisując to w swoim kodzie w taki sposób jak byśmy chcieli żeby to działało po naszej stronie. Dodam tylko, że jest to sposób żeby rozpocząć development wcześniej, a nie dopiero jak dostaniemy gotową/działającą funkcjonalność, o którą musimy się oprzeć.

Przykład z życia

Często pokazywanym przykładem z życia przy wykorzystaniu tego rozwiązania są przejściówki w kablach. Wyobraź sobie, że mamy wejście USB-C, natomiast kabel, którym dysponujemy to USB-A. Tutaj wrzucam link z grafikami porównującymi wspomniane standardy USB. W takiej sytuacji potrzebujemy przejściówki, czyli adaptera. Kolejnym przykładem może być zamiana jednostek z km/h na MPH. Kolejny przykład jaki możemy wymyślić to klucze (narzędzia), gdzie możemy potrzebować skorzystać z innej nasadki niż posiadamy i będziemy potrzebować przejściówki.

Rozwiązanie

Rozwiązanie skupia się wokół zaprojektowania klasy **wrappera**, tzw. opakowania. Wrapper inaczej jest nazywany **Adapterem**. Adapter taki ma komunikować ze sobą obiekty adaptowane (czyli te "oryginalne", "źródłowe"), które chcemy dopasować. Adapter taki jednocześnie odwzorowuje klasę klienta (czyli klasę, która z tego adaptera korzysta) na klasę adaptowaną.

Przykład w kodzie

Przykład w kodzie zastosujemy aby napisać adapter z Cali do Metrów. Przykład jaki pokażemy to telewizor Sony TV, który ma 49'. Postaramy się wyrazić tę wielkość w metrach. Mówiąc o rozmiarze telewizora cały czas będziemy mówić o rozmiarze jego przekątnej.

Interface Dimension

```
public interface Dimension {

    BigDecimal getDimension();
}
```

Interface `Dimension` odpowiada za zwracanie rozmiaru telewizora. Oryginalnie rozmiar telewizora jest podawany w calach, np. 49'.

Klasa SonyTV

```
public class SonyTV implements Dimension {
    @Override
    public BigDecimal getDimension() {
        return BigDecimal.valueOf(49);
    }
}
```

W tym momencie jeżeli chcemy zastosować wzorzec **Adapter**, możemy stworzyć interface **DimensionAdapter**, który również będzie zawierał metodę `getDimension()`.

Interface DimensionAdapter

```
public interface DimensionAdapter {

    BigDecimal getDimension();
```

```
}
```

Oraz jego implementację:

Klasa DimensionAdapterImpl

```
@Data
@AllArgsConstructor
public class DimensionAdapterImpl implements DimensionAdapter {

    private static final double INCH_TO_METER = 0.0254;

    private Dimension dimension;

    @Override
    public BigDecimal getDimension() {
        return convertIntToMeter(dimension.getDimension());
    }

    private BigDecimal convertIntToMeter(BigDecimal inch) {
        return BigDecimal.valueOf(INCH_TO_METER).multiply(inch);
    }
}
```

Jeżeli teraz będziemy chcieli wywołać napisany kod, możemy to zrobić w ten sposób:

Klasa TVRunner

```
public class TVRunner {

    public static void main(String[] args) {
        Dimension dimension = new SonyTV();
        DimensionAdapter dimensionAdapter = new DimensionAdapterImpl(dimension);
        System.out.println("original: " + dimension.getDimension());
        System.out.println("adapted: " + dimensionAdapter.getDimension());
    }
}
```

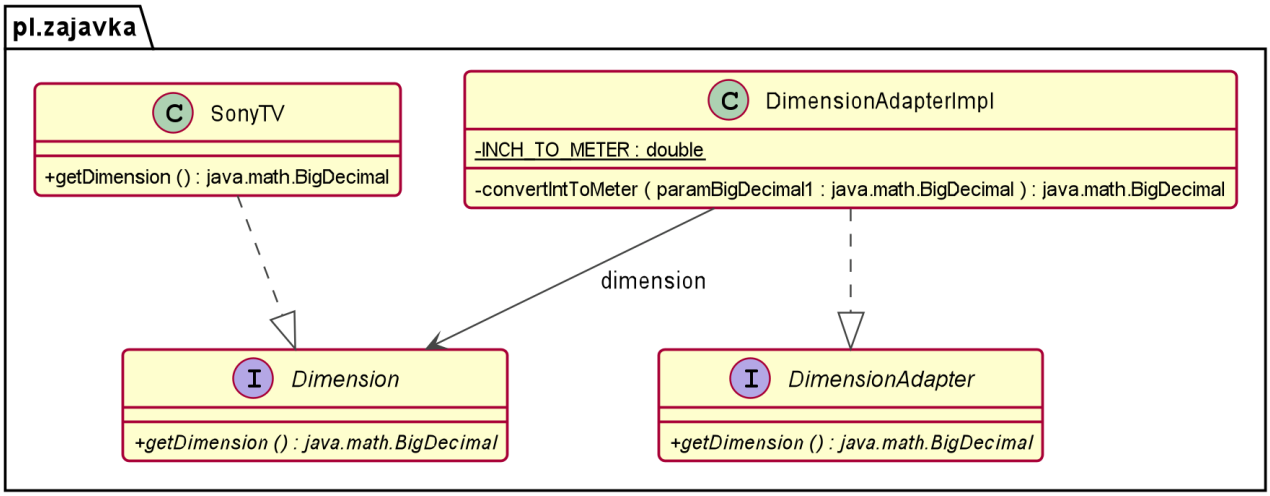
Zalety tego podejścia:

- Jeżeli pracujemy nad bardzo starym kodem i chcemy go wymienić, to zamiast modyfikować go od razu możemy robić taką modyfikację na 3 kroki.
    - Napisać Adapter ze starego kodu do nowego, który pozwoli nam nadal korzystać ze starej funkcjonalności, ale kod korzystający będzie już się opierał o kod "po nowemu",
    - Napisać na nowo implementację, która nas interesuje,
    - Przepiąć kod adaptowany z tego starego na nowy
- Jeżeli będziemy musieli zintegrować się z API, którego jeszcze nie mamy, to możemy najpierw napisać kod Adaptera, który będzie zwracał nam jakieś dane na sztywno. Następnie wykorzystać ten kod adaptera do napisania kodu naszej aplikacji. Następnie gdy API, z którego mamy korzystać zostanie ukończone, możemy zmodyfikować adapter aby korzystał z prawidłowego źródła.
- Wcześniej wspomnieliśmy również o dostosowaniu istniejącego juz API do naszej aplikacji jeżeli nie

spełnia ono naszych wymagań.

Diagram UML

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



Obraz 1. UML Adapter Class Diagram

Composite

Problem jaki rozwiązuje

Jeżeli mamy rodzinę obiektów, które mają być potraktowane w naszym kodzie dokładnie w ten sam sposób - zastosujemy wzorzec **Composite** (kompozyt).

Przykład z życia

Wyobraź sobie, że pracujemy w ZOO i każde zwierze ma zostać nakarmione. Oczywiście gdy piszemy kod, modelujemy świat na swój sposób, więc możemy założyć, że mamy rodzinę zwierząt i w momencie, gdy chcemy je nakarmić, wystarczy wywołać metodę `hereIsTheFood()`, a one będą już wiedziały co z tym zrobić. Takie rozwiązanie postaramy się pokazać w kodzie.

Rozwiązanie

Wyróżnimy 3 rodzaje obiektów, aby zaimplementować ten wzorzec:

- **Base component** - Komponent bazowy - będzie to interfejs, który będzie reprezentował każde zwierzątko w ZOO. Wywołując kod, który nakarmi zwierzątka, wywołamy metodę z tego interfejsu. Oczywiście równie dobrze może to być klasa abstrakcyjna.
- **Leaf** - liść - obiekty, które zdefiniują zachowania poszczególnych zwierzątek. Zwierzątko będzie implementowało **base component**. Zwierzęta nie wiedzą też nic o sobie wzajemnie.
- **Composite** - kompozyt - czyli klasa, która będzie zawierała w sobie kolekcję liści i wywoływała

metody z każdego z nich.

Przykład w kodzie

Aby móc zaimplementować powyżej opisaną sytuację, stwórzmy interface `Animal` oraz klasę `Food`.

Interface Animal

```
public interface Animal {

    void eat(Food food);
}
```

Klasa Food

```
public class Food {}
```

Następnie stwórzmy 3 różne implementacje interfejsu `Animal`, czyli `Monkey`, `Elephant` oraz `Tiger`.

Klasa Monkey

```
public class Monkey implements Animal {

    @Override
    public void eat(final Food food) {
        System.out.println("Monkey eating food: " + food);
    }
}
```

Klasa Tiger

```
public class Tiger implements Animal {

    @Override
    public void eat(final Food food) {
        System.out.println("Tiger eating food: " + food);
    }
}
```

Klasa Elephant

```
public class Elephant implements Animal {

    @Override
    public void eat(final Food food) {
        System.out.println("Elephant eating food: " + food);
    }
}
```

Powyższe klasy `Monkey`, `Elephant` oraz `Tiger` są Liśćmi w nomenklaturze, która została opisana wcześniej. Napiszmy zatem teraz klasę `ZOO`:

```
@Data
@AllArgsConstructor
public class ZOO implements Animal {

    private final List<Animal> animals = new ArrayList<>();

    @Override
    public void eat(Food food) {
        animals.forEach(animal -> animal.eat(food));
    }

    public void add(Animal animal) {
        this.animals.add(animal);
    }

    public void remove(Animal animal) {
        this.animals.remove(animal);
    }

    public void clear() {
        System.out.println("Removing all animals");
        this.animals.clear();
    }

}
```

Klasa `ZOO` przechowuje wszystkie zwierzęta i jest w stanie rozdać jedzenie wszystkim z nich w jednakowy sposób. Zaimplementowaliśmy w niej również interface `Animal`, aby miała ona zdefiniowaną metodę do karmienia, dokładnie taką samą jak każde zwierze osobno. Wykorzystanie klasy `ZOO` może wyglądać następująco:

```
public class RunAnimalComposite {

    public static void main(String[] args) {
        ZOO zoo = new ZOO();
        zoo.add(new Monkey());
        zoo.add(new Elephant());
        zoo.add(new Tiger());
        zoo.add(new Tiger());
        zoo.eat(new Food());
    }
}
```

W powyższym przykładzie możemy nakarmić całe zoo z jednego miejsca, gdyż każde zwierze ma ustandaryzowany sposób jedzenia. Inaczej mówiąc, tworząc taką rodzinę obiektów, możemy zapewnić pewien **kontrakt** mówiący, że każde z nich będzie musiało być w stanie obsłużyć jakąś konkretną sytuację - tutaj akurat jest to jedzenie. Widzisz jednocześnie genialne zastosowanie **polimorfizm**u? Na ekranie drukuje się wynik podobny do tego poniżej, gdyż klasa `Food` nie ma zdefiniowanej metody `toString()`.

```
Monkey eating food: pl.zajavka.Food@13969fbe
Elephant eating food: pl.zajavka.Food@13969fbe
Tiger eating food: pl.zajavka.Food@13969fbe
```

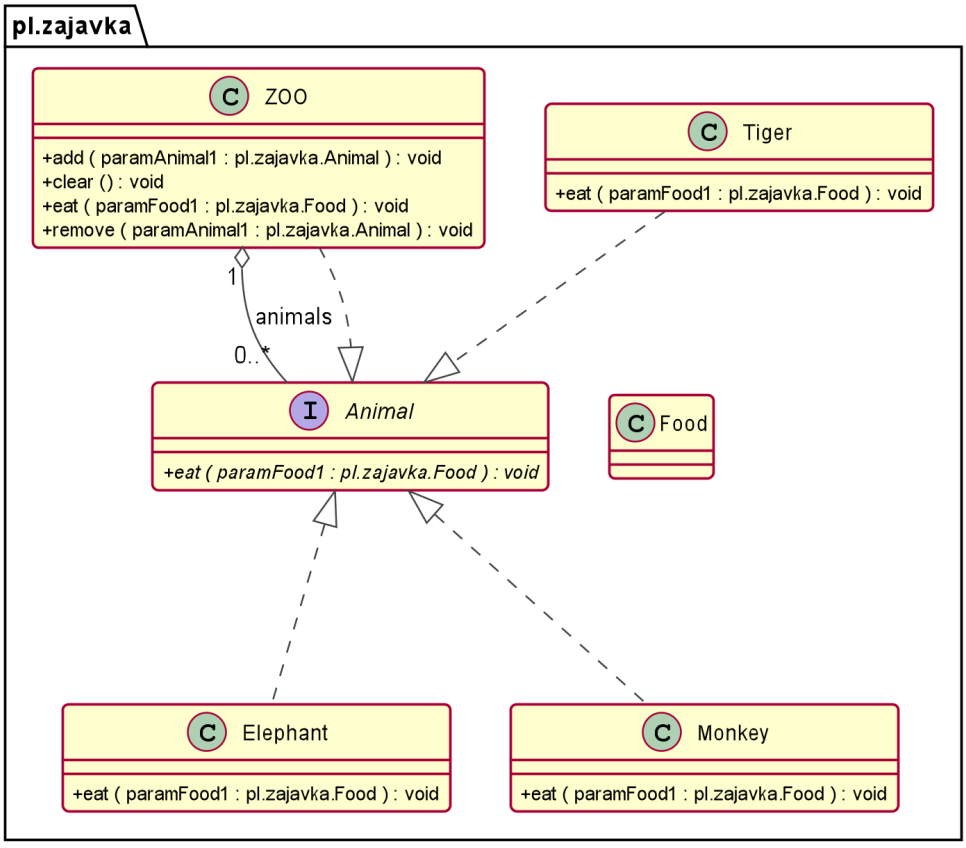
```
Tiger eating food: pl.zajavka.Food@13969fbe
```

Zalety tego podejścia:

- **Composite** pozwala nam w jednakowy sposób traktować całą rodzinę obiektów. Należy jednak wtedy w praktyce dobrze dobierać elementy do takiej rodziny, żeby nie okazało się, że jakieś obiekty są dodane na siłę i nie są w stanie implementować interfejsu **base component**.
- Poszczególne klasy **liście**, czyli np. `Monkey`, `Elephant` oraz `Tiger` nie wiedzą, że są elementem kompozytu, więc możemy w nich nadal dodawać logikę specyficzną dla tylko tej jednej danej klasy.

Diagram UML

Aby pomóc zwizualizować i zapamiętać ten wzorzec, poniżej znajdziesz diagram klas, który obrazuje przykład pokazany w kodzie. Na diagramie nie są pokazane dowiązania do klasy `Food`, żeby nie zaciemnić obrazka.



Obraz 2. UML Composite Class Diagram

Facade

Problem jaki rozwiązuje

Czasami może przytrafić się nam sytuacja, gdzie będziemy mieli zestaw kilku interfejsów i będziemy chcieli je w jakiś sposób uspójnić, zunifikować. Inaczej można to ująć takimi słowami, że mając zestaw kilku interfejsów będziemy chcieli zapewnić do nich dostęp z jednego punktu - **Fasady**. Przyjmuje się, że

interfejsy, na których będziemy budować **fasadę** powinny zapewniać podobne funkcjonalności. Tzn. Nie będziemy próbować budować **fasady** nad interfejsami, które definiują metody do wsiadania do samochodu i do picia mleka. (Musiałem wymyślić coś abstrakcyjnego 😄) Jeszcze inaczej można powiedzieć, że **Fasada** służy do tego aby ukryć złożoność jakiegoś systemu za pojedynczym punktem dostępu. Oczywiście jeżeli potrzebujemy skorzystać z któregoś z przykrywanych interfejsów - nadal jest to możliwe.

Przykład z życia

Przyjmijmy, że chcemy przykryć za fasadą proces wytworzenia samochodu. Proces taki składa się z wielu kroków, ale załóżmy, że chcemy te kroki zgrupować ze sobą i przykryć fasadą.

Rozwiązanie

Aby zbudować fasadę, musimy stworzyć klasę, która zgrupuje ze sobą wywołanie kilku interfejsów i sama udostępni metodę, która pozwoli nam te interfejsy wywołać.

Przykład w kodzie

Załóżmy, że proces wyprodukowania samochodu będzie się składał z następujących kroków (każdy kto się na tym zna, proszę się nie obrażać, skupiamy się na kodzie, a nie na poprawnym oddaniu procesu montażu auta):

Umowny proces produkcji auta

```
frameAssembler.assembleFrame(); // złożenie ramy
painter.paintSkeleton(); // malowanie szkieletu
engineMaker.fastenEngine(); // przymocuj silnik
cockpitAssembler.mountCockpit(); // zamontuj kokpit
doorMaker.installDoor(); // zamontuj drzwi
seatsMaker.attachSeats(); // zamontuj siedzenia
wheelsProducer.attachWheels(); // przykręć koła
```

Możemy na tej podstawie stworzyć jedną metodą `produceCar()`, która będzie chowała w sobie całą logikę tworzenia samochodu i jedyne co będziemy musieli zrobić aby taki samochód stworzyć to wywołać metodę `produceCar()`. Przykład (nie podaję kodu poszczególnych klas, bo nie jest on w tym przykładzie potrzebny):

Klasa CarFactoryFacade

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CarFactoryFacade {
    private FrameAssembler frameAssembler;
    private Painter painter;
    private EngineMaker engineMaker;
    private CockpitAssembler cockpitAssembler;
    private DoorMaker doorMaker;
    private SeatsMaker seatsMaker;
    private WheelsProducer wheelsProducer;
```

```
    public void produceCar() {
        frameAssembler.assembleFrame(); // złożenie ramy
        painter.paintSkeleton(); // malowanie szkieletu
        engineMaker.fastenEngine(); // przymocuj silnik
        cockpitAssembler.mountCockpit(); // zamontuj kokpit
        doorMaker.installDoor(); // zamontuj drzwi
        seatsMaker.attachSeats(); // zamontuj siedzenia
        wheelsProducer.attachWheels(); // przykręć koła
    }
}
```

Jak widać cała skomplikowana logika produkcji auta została zamknięta w jednej klasie, dzięki czemu możemy z innych miejsc w kodzie wywoływać tylko jedną metodę `produceCar()`.

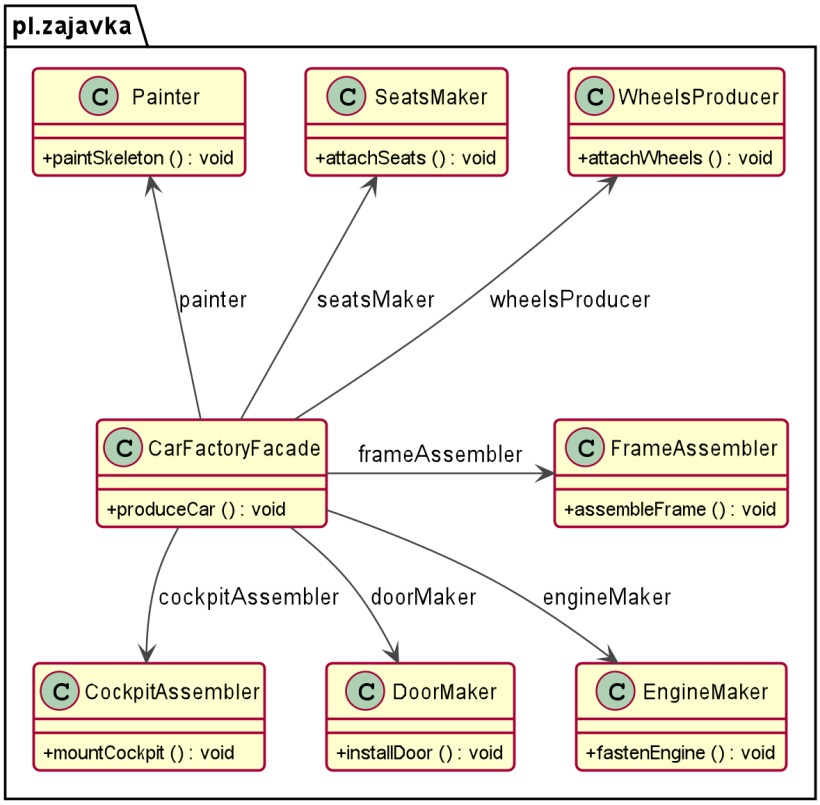
Zalety tego podejścia:

- Umieszczamy logikę w jednym źródle prawdy, czyli w jednym miejscu, w którym wiemy, że mamy umieszczone kroki do ukończenia pewnego skomplikowanego zadania. Daje to porządek w kodzie, nie musimy tej logiki wtedy powielać w innych miejscach.
- Nie blokuje nam to użycia wspomnianych klas `FrameAssembler`, `SeatsMaker` itp. w innych miejscach w kodzie - możemy z nich nadal korzystać.

Z tym wzorcem należy jednak zachować rozsądek i wyważyć złoty środek. Tzn. nie wkładać go na siłę wszędzie gdzie się da ☺.

Diagram UML

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



Obraz 3. UML Facade Class Diagram

Proxy

Problem jaki rozwiązuje

Wzorzec ten będzie nam potrzebny jeżeli będziemy zastanawiać się nad zapewnieniem kontrolowanego dostępu do jakiegoś zasobu. Czyli np. mamy fragment kodu, który może zostać wykonany pod warunkiem, że użytkownik, który go wykonuje ma odpowiednie uprawnienia.

Innym przykładem zastosowania **Proxy** może być sytuacja gdy wiemy, że często odpytujemy jakiś zasób o tę samą informację. Czyli wyobraź sobie, że co 5 sekund pytasz jakiś serwis internetowy o pogodę, a jednocześnie wiesz, że dane na tej stronie internetowej zmieniają się co 24h. Możemy wtedy wprowadzić obiekt proxy, który będzie pośredniczył w odpytaniu serwisu pogodowego o dane i będzie pamiętał, że następna zmiana nastąpi dopiero za np. 5 godzin. Wtedy **Proxy** również wie, że nie ma sensu odpytywać serwisu internetowego o pogodę, bo przecież odpowiedź i tak jest taka sama, więc może równie dobrze zwrócić to co zwrócił nam ostatnio. Obiekt, który przetrzymuje takie informacje w ten sposób w praktyce nazywa się **cache**, natomiast sam proces **cacheowaniem**. Czyli jeżeli przeczytasz gdzieś o **cacheowaniu** to wiesz już, że chodzi o to, żeby zapisać gdzieś informacje, które wiesz, że się nie zmienią, żeby nie musieć cały czas odpytywać źródła.

W praktyce spotkasz się z takim stwierdzeniem jak **Proxy Server** - link. Server taki jest pośrednikiem w przekazywaniu żądań pomiędzy maszynami. Może on wprowadzać dodatkową warstwę zabezpieczeń gdy zależy nam na tym, aby dostęp do lub z jakiejś strony internetowej został zablokowany.

Przykład z życia

Możemy wrócić do przykładu z Pizzami i napisać kod, który pozwala nam na wypieczenie pizzy, ale informacja do kucharza trafia przez obiekt **Proxy** (takim proxy może być np. kelner). Innymi słowy, ktoś na recepcji wie już, że jeżeli jest to pizza, której kucharz się nie podejmie (np. Hawajska 😁) to bez sensu jest mu nawet o tym wspominać bo i tak tego nie zrobi. W takiej sytuacji już obiekt **Proxy** może odpowiedzieć, że nie pieczemy tutaj takiej pizzy.

Rozwiązanie

W tym celu wprowadzamy obiekt **Proxy**, który implementuje interesujący nas interface i jednocześnie jest on od niego zależny przy wykorzystaniu kompozycji. W ten sposób obiekt **Proxy** spełnia kontrakt (bo implementuje interface) obiektu, do którego ma przekazać informacje. Jednocześnie ma dodany ten obiekt przy wykorzystaniu kompozycji aby przekazać mu informacje po wstępnej weryfikacji. Już pokazujemy kod!

Przykład w kodzie

Interface PizzaBaker

```
public interface PizzaBaker {

    void bake(String pizza);
}
```

Dodajemy do tego implementację tego interfejsu, czyli faktycznego pracownika wypiekającego pizzę:

Klasa PizzaBakerImpl

```
public class PizzaBakerImpl implements PizzaBaker {
    @Override
    public void bake(String pizza) {
        System.out.println("Baking pizza: " + pizza);
    }
}
```

Aby zapewnić kontrolowany dostęp do piekarza należy wykorzystać obiekt **Proxy**, który będzie kontrolował dostęp do piekarza. W tym celu wprowadzamy klasę **PizzaBakerProxy**:

Klasa PizzaBakerProxy

```
@RequiredArgsConstructor
public class PizzaBakerProxy implements PizzaBaker {

    private static final List<String> DENIED_PIZZAS = List.of("Hawaiian");

    private final PizzaBaker executor;

    @Override
    public void bake(String pizza) {
        if (DENIED_PIZZAS.contains(pizza)) {
```

```
            throw new RuntimeException(String.format("%s? We don't do this here!", pizza));
        } else {
            executor.bake(pizza);
        }
    }
}
```

Jeżeli teraz będziemy chcieli wykorzystać ten wzorzec w kodzie, możemy zrobić to w ten sposób:

```
public class ProxyRunner {

    public static void main(String[] args){
        PizzaBaker executor = new PizzaBakerProxy(new PizzaBakerImpl());
        try {
            executor.bake("Pepperoni");
            executor.bake("Hawaiian");
            executor.bake("Margeritta");
        } catch (Exception e) {
            System.err.printf("Exception: %s", e.getMessage());
        }
    }

}
```

Na ekranie zostanie wydrukowane:

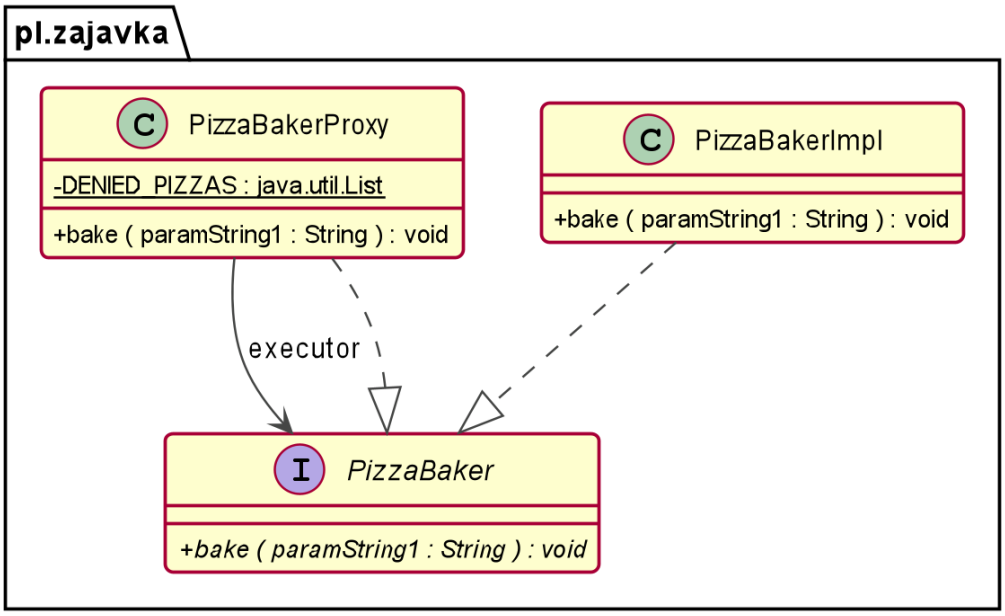
```
Baking pizza: Pepperoni
Exception: Hawaiian? We don't do this here!
```

Zalety tego podejścia:

- Możemy kontrolować dostęp do jakiegoś zasobu i dokonywać wstępnej weryfikacji na wcześniejszym poziomie.
- Możemy zmniejszyć czas odpowiedzi na jakieś zapytanie, jeżeli obiekt **Proxy** wie, że odpowiedź w danym przedziale czasu nie ulegnie zmianie. Wtedy sam obiekt **proxy**, bez odpytywania obiektu źródłowego może taką odpowiedź zwrócić.

Diagram UML

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



Obraz 4. UML Proxy Class Diagram

Decorator

Problem jaki rozwiązuje

Jak sama nazwa wskazuje, wzorzec ten służy do tego aby dekorować. Czyli dodać do obiektu jakieś dekoracje - bombki, świeczki, cukierki 😜.

Kiedy taka sytuacja może nam się przytrafić? Jeżeli będziemy korzystać z jakiejś biblioteki, która będzie nam dostarczała obiekty i będziemy potrzebowali taki obiekt zmodyfikować. Zmodyfikować, czyli dodać do niego w naszym kodzie jakieś pola albo metody, a jednocześnie nie chcemy modyfikować samego kodu źródłowego klasy takiego obiektu.

Pamiętajmy, że jeżeli korzystamy z bibliotek, to nie możemy modyfikować kodu biblioteki w prosty sposób. Wynika to z tego, że biblioteka jest najczyściej dostarczana w formie pliku `.jar`. Musielibyśmy zatem pobrać kod źródłowy takiej biblioteki, wtedy go modyfikować i dołączyć do naszego projektu. Jeżeli wtedy pobralibyśmy kolejną wersję takiej biblioteki to ponownie musielibyśmy modyfikować ten kod źródłowy (bo pobraliśmy nowy w nowszej wersji). Po co się bawić w takie rzeczy, jak jest **Decorator**.

Należy tutaj zaznaczyć, że **Decorator** może być napisany w taki sposób, że możemy nadal korzystać z oryginalnej wersji obiektu, albo z tej udekorowanej.

Przykład z życia

Wyobraźmy sobie, że mamy kupiony samochód i chcemy ten konkretny samochód (a nie wszystkie możliwe samochody) udekorować poprzez dodanie spoilera, dodanie neonów albo wymianę felg. Należy jednak pamiętać, że są do dodatki do obiektu, nie modyfikujemy oryginalnego stanu obiektu. Czyli patrząc przez analogię, jak dodajemy spoiler to nie zabieramy drzwi z samochodu.

Rozwiązanie

W pierwszym kroku musimy zdefiniować interface lub klasę abstrakcyjną np. `Car`, która zdefiniuje metodę np. `create()`. Następnie implementujemy tę metodę np. w klasie `CarImpl`. Implementacja ta służy jako baza, którą będziemy dekorować. Następnie tworzymy klasę abstrakcyjną np. `CarDecorator`, która jednocześnie implementuje interface `Car` oraz wykorzystuje go przy pomocy kompozycji. Jeżeli teraz stworzymy konkretne dekoratory, np. `RimsDecorator`, `SpoilerDecorator`, to każdy z nich będzie w stanie dodać dodatkową cechę do obiektu. Przejdźmy natomiast do przykładu.

Przykład w kodzie

Na początek zdefiniujmy interface `Car`.

Interface Car

```
public interface Car {
    String create();
}
```

Następnie zdefiniujmy klasę, która implementuje ten interface, czyli `CarImpl`. To właśnie tę klasę będziemy dekorować.

Klasa CarImpl

```
public class CarImpl implements Car {

    @Override
    public String create() {
        return "Car";
    }
}
```

Teraz możemy przejść do zdefiniowania klasy abstrakcyjnej `CarDecorator`. Klasa ta jednocześnie implementuje interface `Car` oraz korzysta z niego przy wykorzystaniu kompozycji.

Klasa CarDecorator

```
@AllArgsConstructor
public abstract class CarDecorator implements Car {

    private final Car car;

    @Override
    public String create() {
        return car.create();
    }
}
```

Teraz możemy określić konkretne dekoratory, czyli np. `NeonDecorator`, `SpoilerDecorator`, `RimsDecorator`.

Klasa NeonDecorator

```
public class NeonDecorator extends CarDecorator {

    public NeonDecorator(Car car) {
        super(car);
    }

    @Override
    public String create() {
        // możemy również dodać tutaj wywołanie przed metodą super.create()
        String created = super.create();
        // możemy również dodać tutaj wywołanie po metodzie super.create()
        return created + withNeon();
    }

    private String withNeon() {
        return " with " + this.getClass().getSimpleName();
    }
}
```

Jak zostało to pokazane w komentarzach w kodzie w klasie `NeonDecorator`, możemy również dodać logikę przez i po wywołaniu metody `super.create()`.

Klasa SpoilerDecorator

```
public class SpoilerDecorator extends CarDecorator {

    public SpoilerDecorator(Car car) {
        super(car);
    }

    @Override
    public String create() {
        return super.create() + withSpoiler();
    }

    private String withSpoiler() {
        return " with " + this.getClass().getSimpleName();
    }
}
```

Klasa RimsDecorator

```
public class RimsDecorator extends CarDecorator {

    public RimsDecorator(Car car) {
        super(car);
    }

    @Override
    public String create() {
        return super.create() + withRims();
    }

    private String withRims() {
        return " with " + this.getClass().getSimpleName();
```

```
    }
}
```

Zwróć uwagę na wielokrotne wywołanie metody `super.create()`. Wynika to z tego, że musimy wywołać metody z obiektów bazowych, bo przecież musimy mieć bazę do dekorowania ☺.

Jeżeli teraz chcemy stworzyć obiekt klasy `CarImpl`, nie ma żadnego problemu żeby normalnie go stworzyć. Jeżeli natomiast chcielibyśmy stworzyć teraz obiekt klasy `CarImpl`, który zostałby udekorowany spoilerem, musimy do tego wykorzystać klasę `SpoilerDecorator`, która w wywołaniu konstruktora przyjmie obiekt implementujący interface `Car`. Poniżej przykład:

Klasa DecoratorRunner

```
public class DecoratorRunner {

    public static void main(String[] args) {
        Car car1 = new SpoilerDecorator(new CarImpl());
        System.out.println(car1.create());

        Car car2 = new SpoilerDecorator(new RimsDecorator(new CarImpl()));
        System.out.println(car2.create());

        Car car3 = new SpoilerDecorator(new NeonDecorator(new RimsDecorator(new CarImpl())));
        System.out.println(car3.create());
    }
}
```

Dzięki powyższemu zapisowi, na ekranie zostanie wydrukowane:

```
Car with Spoiler
Car with Rims with Spoiler
Car with Rims with Neon with Spoiler
```

Stosując to podejście mogliśmy stworzyć obiekt klasy `Car`, który został udekorowany tylko spoilerem, albo obiekt klasy `Car`, który został udekorowany spoilerem oraz nowymi felgami.

Czyli podsumowując, **Decorator** pozwala nam dodawać nowe właściwości lub zachowania obiektom, poprzez opakowywanie ich w inne obiekty, które definiują konkretne właściwości lub zachowania. Ważne jest to, że nadal możemy korzystać z oryginalnego obiektu, dekoracje nie są propagowane automatycznie na inne obiekty. Możemy jednak korzystać jednocześnie z obiektów dekorowanych.

Można powiedzieć, że **Decorator** może być alternatywą dla dziedziczenia. W przypadku dziedziczenia cechy są współdzielone przez wszystkie stworzone obiekty. Tutaj natomiast możemy to zrobić selektywnie. Różnica jest też taka, że dziedziczenie dodaje swoje zachowania na etapie kompilacji kodu źródłowego. Pokazane podejście dodaje natomiast zachowania w trakcie działania programu (w runtime). Zobaczyliśmy też, że kolejne dekoracje można zagłębiać, nie ma tutaj ograniczeń.

Zalety tego podejścia:

- Wzorzec ten pozwala na modyfikację obiektów w elastyczny sposób,
- Możemy na jeden obiekt nałożyć wiele dekoracji, nie ogranicza się to tylko do jednej.

Dekorujemy zewnętrzną bibliotekę

Rozpatrzmy teraz przypadek, w którym chcemy udekorować klasę, która jest dostarczana przez bibliotekę. Za przykład weźmy klasę `Attribute` z biblioteki `jsoup`. W tym celu dodajmy tę zależność do projektu i napiszmy poniższy przykład.

Klasa AttributeDecorator

```
import org.jsoup.nodes.Attribute;

public abstract class AttributeDecorator extends Attribute {

    private final Attribute attribute;

    public AttributeDecorator(final Attribute attribute) {
        super(attribute.getKey(), attribute.getValue());
        this.attribute = attribute;
    }

    @Override
    public String getKey() {
        return super.getKey();
    }
}
```

Zwróć uwagę, że tym razem zamiast implementować interface `Car`, rozszerzamy klasę `Attribute`. Chcemy udekorować metodę `getKey()`, więc musimy ją nadpisać i wywołać implementację z klasy bazowej `Attribute`. Robimy to przez wywołanie metody `super.getKey()`.

Pokazywane podejście zadziała tylko jeżeli klasa, którą rozszerzamy (tutaj `Attribute`) **nie jest final** i jednocześnie metoda, którą będziemy dekorować również **nie jest final**. Jeżeli którekolwiek będzie final będziemy musieli poszukać innego podejścia.

Następnie dodajmy dwa dekoratory `AttributeBeforeDecorator` i `AttributeOtherDecorator`.

Klasa AttributeBeforeDecorator

```
import org.jsoup.nodes.Attribute;

public class AttributeBeforeDecorator extends AttributeDecorator {

    public AttributeBeforeDecorator(final Attribute attribute) {
        super(attribute);
    }

    @Override
    public String getKey() {
        System.out.println("Decorator before step");
        return super.getKey();
    }
}
```

Klasa AttributeOtherDecorator

```
import org.jsoup.nodes.Attribute;

public class AttributeOtherDecorator extends AttributeDecorator {

    public AttributeOtherDecorator(final Attribute attribute) {
        super(attribute);
    }

    @Override
    public String getKey() {
        System.out.println("Decorator other step");
        return super.getKey();
    }
}
```

Klasy te dekorują klasę `Attribute` w ten sposób, że dodają pewną logikę (tutaj akurat drukowane napisu na ekranie), która zostanie wywołana przed wywołaniem metody `getKey()`, z klasy `Attribute`. Jeżeli teraz będziemy chcieli taki kod uruchomić, to możemy zrobić to przykładowo w ten sposób:

Klasa DecoratorRunner

```
import org.jsoup.nodes.Attribute;

public class DecoratorRunner {

    public static void main(String[] args) {
        System.out.println("Example 1");
        Attribute attribute1 = new AttributeBeforeDecorator(
            new Attribute("key1", "value1"));
        System.out.println(attribute1.getKey());

        System.out.println();
        System.out.println("Example 2");
        Attribute attribute2 = new AttributeBeforeDecorator(
            new AttributeOtherDecorator(
                new Attribute("key2", "value2")));
        System.out.println(attribute2.getKey());
    }
}
```

Po uruchomieniu kodu powyżej, na ekranie zostanie wydrukowane:

```
Example 1
Decorator before step
key1

Example 2
Decorator other step
Decorator before step
key2
```

Widzimy zatem, że każdy kolejny dekorator, który opakowuje poprzedni dekorator, jest wywoływany po dekoratorze, który go poprzedza. Czyli przy kolejności:

```
new AttributeBeforeDecorator(new AttributeOtherDecorator(new Attribute("key2", "value2")));
```

Na ekranie jest drukowane najpierw `Decorator other step`, a następnie `Decorator before step`.

Dokładamy Java 8

Jeżeli dobrze czujesz już klimat interfejsów funkcyjnych, widzisz też pewnie, że w przypadku przykładu z `Car` i `CarDecorator` można zaimplementować ten wzorzec przy wykorzystaniu **lambdy**. Zostawmy w tym celu interface `Car` oraz klasę `CarImpl` w takiej postaci w jakiej były wcześniej.

Interface Car

```
public interface Car {
    String create();
}
```

Klasa CarImpl

```
public class CarImpl implements Car {

    @Override
    public String create() {
        return "Car";
    }
}
```

To gdzie ta lambda? Poniżej.

Klasa DecoratorRunner

```
public class DecoratorRunner {

    public static void main(String[] args) {
        CarImpl originalCar = new CarImpl();

        Car car1 = () -> "Before car! " + originalCar.create() + " with Spoiler";
        System.out.println(car1.create());

        Car car2 = () -> "Before car! " + originalCar.create() + " with Rims" + " with Spoiler";
        System.out.println(car2.create());

        Car car3 = () -> "Before car! " + originalCar.create() + " with Rims with Spoiler with Neon";
        System.out.println(car3.create());
    }
}
```

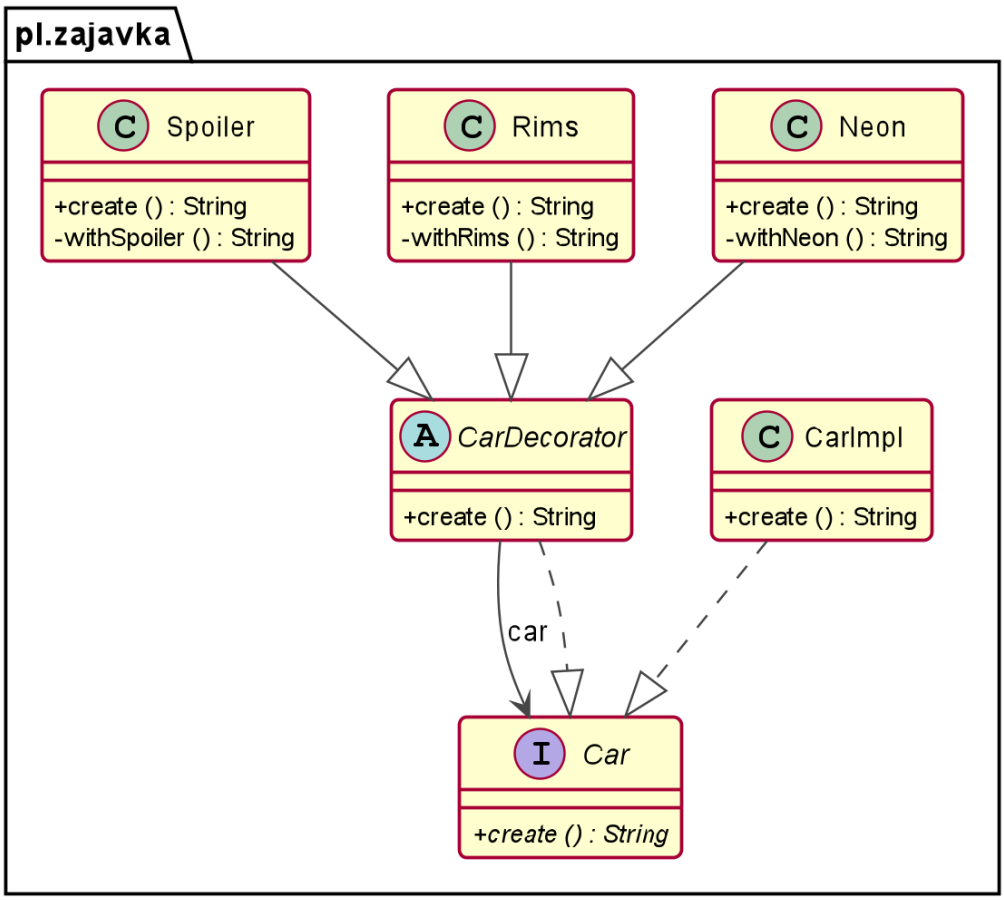
Jeżeli teraz uruchomimy ten kod to na ekranie zobaczymy:

```
Before car! Car with Spoiler
Before car! Car with Rims with Spoiler
Before car! Car with Rims with Spoiler with Neon
```

Powyższy przykład pokazuje, że nie ma konieczności definiowania klas takich jak np. `NeonDecorator`, `SpoilerDecorator`, `RimsDecorator`. Możemy zastąpić implementacje tych klas przy pomocy lambdy. Wynika to z tego, że każda z tych klas w swojej hierarchii dziedziczenia implementuje interface funkcyjny jakim jest `Car`. Jeżeli jednak zależy nam na reużywalności napisanego kodu należy zastanowić się, czy definiowanie oddzielnych klas nie jest lepszym rozwiązaniem.

Diagram UML

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



Obraz 5. UML Decorator Class Diagram

Bridge

Bridge to stworzenie interfejsu będącego mostem (stąd nazwa) pomiędzy dwiema warstwami abstrakcji.

Problem jaki rozwiązuje

Mówiliśmy wcześniej o zasadach takich jak **composition over inheritance** oraz o **DIP**. Znając te zasady wiemy już, że zrobimy sobie mniejsze kuku stosując kompozycję oraz, że powinniśmy w naszych klasach zależeć od abstrakcji (czyli np. interfejsów) zamiast od konkretnych implementacji tych interfejsów. Dzięki temu nasz kod staje się bardziej elastyczny.

Problem jaki jest rozwiązywany tym wzorcem to połączenie ze sobą dwóch abstrakcji, czyli np. klasy

abstrakcyjnej z interfejsem przy wykorzystaniu kompozycji. Łączenie to odbywa się w taki sposób, aby obie te abstrakcje nie widziały szczegółów swojej implementacji.

Przykład z życia

Załóżmy, że mamy dwie struktury obiektów, strukturę figur geometrycznych i strukturę kolorów, na jaki dana figura geometryczna może zostać pomalowana (ten przykład jest bardzo często powielany w internecie ze względu na swoją prostotę). Figura geometryczna niezależnie od tego co to jest za figura chce mieć pole określające jakiego jest koloru. Kolor ten oczywiście również nie jest zapisany na sztywno. Taka figura wie tylko, że ma jakiś kolor. Jaki to będzie kolor okaże się dopiero po uruchomieniu programu.

Rozwiązanie

Stworzymy dwie hierarchie klas. Klasę abstrakcyjną `Shape`, która będzie dziedziczona przez klasy `Square` i `Triangle`. Klasa `Shape` będzie miała zdefiniowane pole `Color`. `Color` natomiast będzie interfejsem, który będzie implementowany przez klasy `Red` oraz `Green`. Relacja pomiędzy klasami `Shape` oraz `Color` jest nazywana mostem (**bridge**). Czyli klasa abstrakcyjna jest zależna od interfejsu.

Przykład w kodzie

Interface Color

```
public interface Color {
    String apply();
}
```

Klasa Shape

```
@ToString
@AllArgsConstructor
public abstract class Shape {
    protected Color color;

    abstract public void draw();
}
```

Możemy teraz zdefiniować konkretne implementacje klasy `Color`, np. `Red` oraz `Green`.

Klasa Red

```
@ToString
public class Red implements Color {

    @Override
    public String apply() {
        return "Color is Red";
    }
}
```

Klasa Green

```
@ToString
public class Green implements Color {

    @Override
    public String apply() {
        return "Color is Green";
    }
}
```

Następnie określamy klasy rozszerzające klasę `Shape`, czyli np. `Square` oraz `Triangle`.

Klasa Square

```
@ToString(callSuper = true)
public class Square extends Shape {

    public Square(final Color color) {
        super(color);
    }

    @Override
    public void draw() {
        System.out.println("Square with color: " + color.apply());
    }
}
```

Klasa Triangle

```
@ToString(callSuper = true)
public class Triangle extends Shape {

    public Triangle(final Color color) {
        super(color);
    }

    @Override
    public void draw() {
        System.out.println("Triangle with color: " + color.apply());
    }
}
```

Jeżeli teraz spróbujemy wywołać ten kod w klasie **BridgeRunner**.

Klasa BridgeRunner

```
public class BridgeRunner {

    public static void main(String[] args) {
        Shape shape1 = new Square(new Red());
        System.out.println(shape1);

        Shape shape2 = new Triangle(new Green());
        System.out.println(shape2);
    }
```

```
}
```

Na ekranie zostanie wydrukowane:

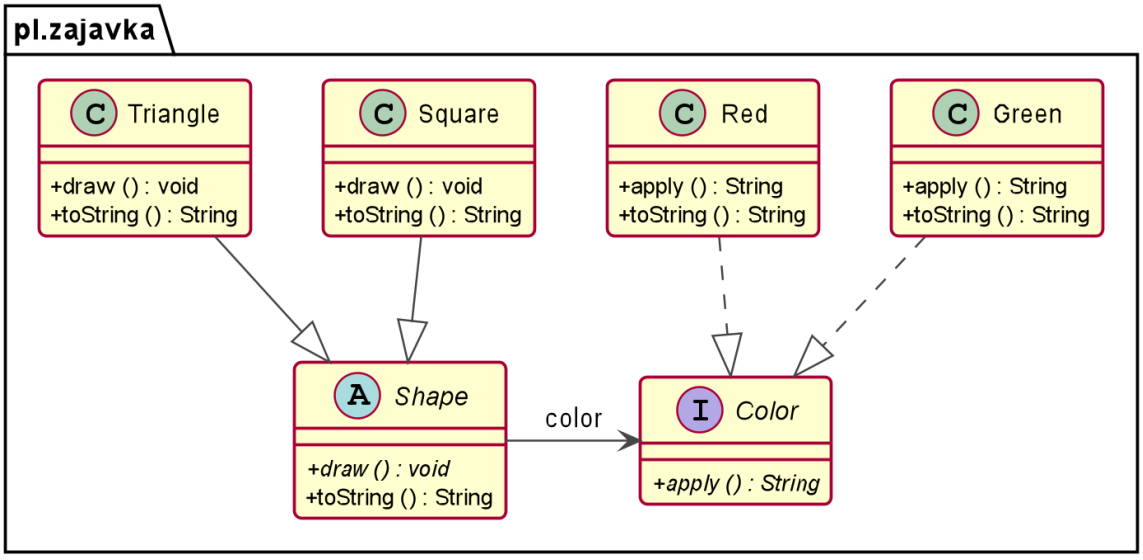
```
Square(super=Shape(color=Red()))
Triangle(super=Shape(color=Green()))
```

Zalety tego podejścia:

- Dzięki stosowaniu tego wzorca uniezależniamy się pomiędzy dwiema strukturami, które ze sobą łączymy.
- Możemy dowolnie zmieniać hierarchię klasy `Shape` oraz hierarchię interfejsu `Color`. Obie te hierarchie nie będą o tym wiedziały wzajemnie.

Diagram UML

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



Obraz 6. UML Bridge Class Diagram

Podsumowanie

Podsumujmy co już wiemy na tym etapie. Wzorce strukturalne określają pewien rodzaj zależności między klasami, który pozwala w znany sposób rozwiązać często powtarzające się problemy, które skupiają się wokół łączenia ze sobą klas, wzbogacania je o jakieś zachowania lub ukrywania pewnych zachowań. We wspomnianych przykładach często należało dodać pewną "warstwę abstrakcji" przed jakimś kodem w celu osiągnięcia spodziewanego efektu. To właśnie o tej dodatkowej warstwie abstrakcji chcę wspomnieć jeszcze raz, żeby zaznaczyć pewną kwestię.

Jeżeli korzystamy z kodu z zewnętrznych bibliotek, albo z kodu z zewnętrznych modułów aplikacji, bardzo dobrą praktyką jest **nie używanie** takiego "zewnętrznego kodu" wszędzie jak popadnie. Dobrze

jest obudować taki kod naszą warstwą abstrakcji i korzystać dopiero z takiej warstwy abstrakcji. Z czego to wynika?

Wyobraź sobie, że napisaliśmy dużo kodu korzystając z jakiejś biblioteki. Kod biblioteki jest używany w 30 naszych klasach. Zmieniamy wersję biblioteki i usunięte zostało z niej w kolejnej wersji dużo funkcjonalności z których korzystamy. Musimy teraz to przepisać w taki sposób, żeby nasz kod nadal działał. I mamy do przerobienia 30 klas..

Wyobraź sobie teraz, że korzystamy z zewnętrznej biblioteki, ale wszystkie jej wywołania są opakowane w **Fasadę**, **Adapter** lub obiekt **Proxy**. I dopiero z tej **Fasady**, **Adaptera** albo obiektu **Proxy** korzystamy w kodzie naszej aplikacji. Dzięki takiemu zabiegowi mamy do zmiany tylko jedną klasę, a nie 30. Jest to skrajny przykład ale obrazuje problem jaki będziemy mieli jeżeli nie opakowujemy "zewnętrznego/cudzego" kodu naszym kodem.