Programowanie aspektowe i wzorce projektowe

05.11.2022

Programowanie aspektowe - wstęp

Po co jest programowanie aspektowe?

Programowanie aspektowe pozwala na:

- Odseparowanie logiki biznesowej od kodu
- Wydzielenie poszczególnych elementów kodu i umieszczenie w oddzielnych aspektach
- Dodawanie kodu wykonywalnego do kodu źródłowego bez zmieniania go

Kompilacja vs wykonywanie

Paradygmat programowania aspektowego realizuje się przez wstrzyknięcie kodu podczas kompilacji lub wykonywania.

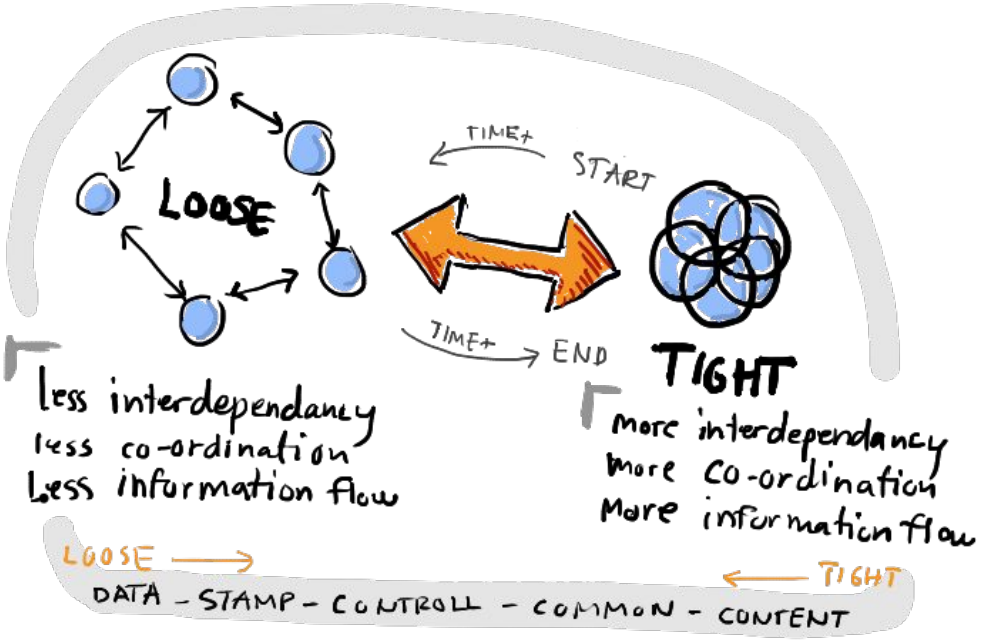
- Realizacja AOP podczas kompilacji powoduje, że otrzymany plik .class różni się od tego, który zostałby stworzony bez framework'a pozwalające na realizację AOP
- Realizacja AOP podczas wykonywania nie zmienia ani pliku .class, ani kodu źródłowego

Interceptor w JEE

Bean - ziarno

- Obiekt klasy zarządzany po stronie kontenera
- Znajduje się w archiwum ziaren
- Nie wymaga dużego nakładu pracy po stronie programisty

Luźne powiązania



Co to jest interceptor?

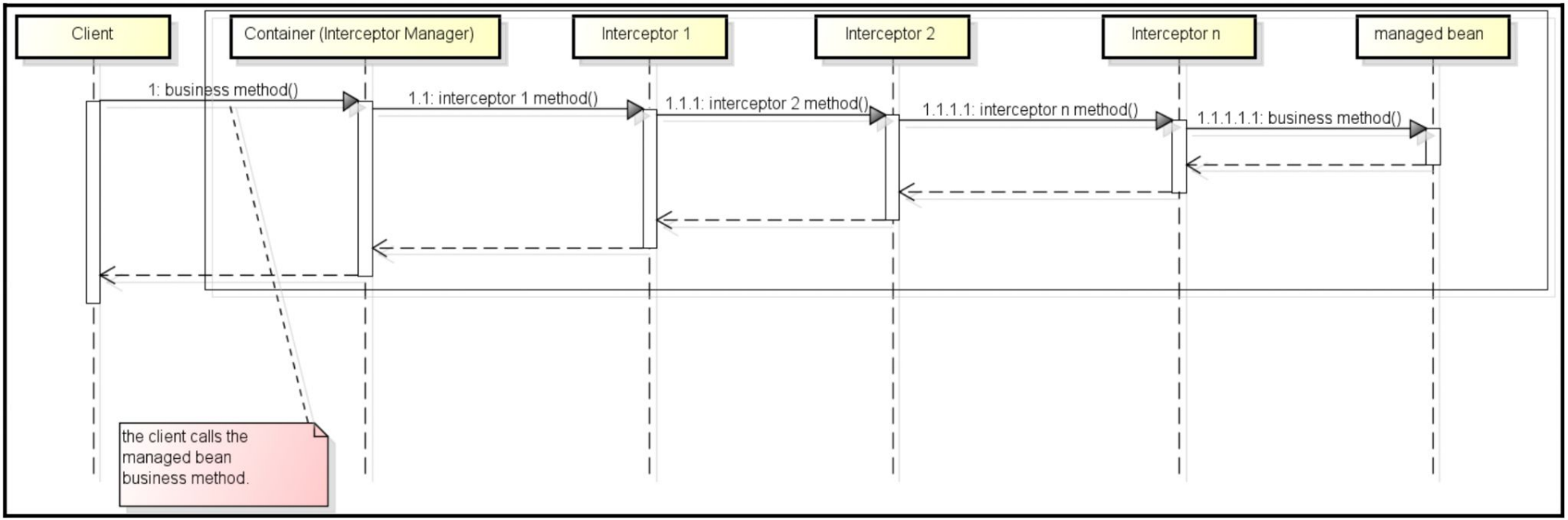
"An interceptor is a class used to interpose in invocation methods or life cycle events that occur in an associated target class."

~ Oracle

Po co jest interceptor?

- Wydziela logikę biznesową
- Tworzy kod, który będzie wywoływany bezpośrednio przed lub po wywołaniu metody
- Przechwytuje wywołań HTTP - działa to jak filtr
- Dekoruje klasy dodatkowymi funkcjami bez wprowadzania zmian w metodzie

Łańcuch interceptorów



Interceptor EJB - implementacja

Interceptor EJB - podstawowe informacje

- Realizacja paradygmatu programowania aspektowego
- Odnosi się do cyklu życia ziaren oraz ich metod biznesowych
- Implementowany w postaci metod:
  - odpowiednio adnotowanych
  - o dowolnym poziomie dostępu (public, package private, protected, private)
  - nie mogących być ani static, ani final
  - deklarowanych bezpośrednio w klasie EJB lub w klasie osobnej

Interceptor EJB - invocation context

Interceptor do skutecznego działania potrzebuje kontekstu wywołania:

- instancję ziarna
- parametry metody
- rezultat wywołania metody

Dane te przez przechowuje obiekt InvocationContext, który jest przekazywany jako parametr do metody interceptora.

Interceptor EJB - invocation context (przykład)

```
@AroundInvoke
public Object aroundInvokeMethod(InvocationContext invocationContext) throws Exception {
    invocationContext.getMethod().getName();
    invocationContext.getMethod().getDeclaringClass();
    invocationContext.getContextData();
    invocationContext.getTarget();
    return invocationContext.proceed();
}
```

Interceptor EJB - klasa interceptorów

- Wystarczy, że posiada w sobie zdefiniowane interceptory
- Nie musi być oznaczona specjalną adnotacją
- Musi posiadać publiczny bezargumentowy konstruktor
- Przypisywana przez adnotację @Interceptors

Interceptor EJB - klasa interceptorów (przykład)

```java
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class DariuszInterceptor {

    public DariuszInterceptor() {}

    @AroundInvoke
    public Object aroundInvokeMethod(InvocationContext invocationContext)
            throws Exception {
        return invocationContext.proceed();
    }

}
```

```java
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;

@Stateless
@Interceptors({DariuszInterceptor.class, DariuszSecondInterceptor.class})
public class TestBean {

    @Interceptors({DariuszInterceptor.class, DariuszSecondInterceptor.class})
    public void businessMethod() {
    }

}
```

Interceptor EJB - metody biznesowe

- Metoda oznaczona adnotacją @AroundInvoke
- Adnotacją można oznaczyć tylko jedną metodę w klasie
- Kolejność ich wykonania zgodna z kolejnością ich deklarowania oraz hierarchią dziedziczenia
- Wykonywane na tym samym stosie wywołania co metoda biznesowa

Interceptor EJB - metody biznesowe (przykład)

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptors;
import javax.interceptor.InvocationContext;

@Interceptors({DariuszInterceptor.class})
public class TestBean {

    @AroundInvoke
    protected Object beanAroundInvokeMethod(InvocationContext ic) throws Exception {
        return ic.proceed();
    }

}
```

Interceptor EJB - cyklu życia ziaren

- Metoda oznaczona jedną z adnotacji:
  - @PostConstruct
  - @PostActivate
  - @PreDestroy
  - @PrePassivate
- Zdefiniowana w klasie ziarna - sygnatura void <METHOD>()
- Zdefiniowana w klasie interceptorów - sygnatura void <METHOD>(InvocationContext)

Interceptor EJB - cyklu życia ziaren (przykład)

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.interceptor.InvocationContext;

    public class DariuszInterceptor {

        public DariuszInterceptor() {}

        @PostConstruct
        private void beanPostConstruct(InvocationContext context) { /* ... */ }

        @PreDestroy
        private void beanPreDestroy(InvocationContext context) { /* ... */ }

    }
```

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;

@Stateless
@Interceptors(DariuszInterceptor.class)
public class TestBean {

    @PostConstruct
    private void beanPostConstruct() { /* ... */ }

    @Interceptors(DariuszInterceptor.class)
    public void businessMethod() {
    }

    @PreDestroy
    private void beanPreDestroy() { /* ... */ }

}
```

Interceptor CDI

Iterceptor CDI - informacje ogólne

- Klasy interceptora zawierają odpowiednie adnotacje
- Dzięki niemu unika się kopiowania wielokrotnie tego samego kodu
- Umożliwia przechwytywanie metod biznesowych czy timeout'ów
- Beany CDI nie zbierają wiedzy na temat implementacji poszczególnych klas, znają jedynie informacje o typie interceptora

Typy wstrzykiwania interceptorów

- @AroundInvoke - przechwytuje podstawowe wywołania metod
- @AroundTimeout - przechwytuje Timery EJB
- @AroundConstruct - przechwytuje metody zwracające jakąś informację zwrotną po skonstruowaniu
- @PostConstruct - przechwytuje zdarzenia po skonstruowaniu
- @PreDestroy - przechwytuje zdarzenia przed zniszczeniem

Przykład implementacji interceptora CDI

```
@Interceptor
public class ExampleInterceptor {
    @AroundInvoke
    public Object exampleInterceptor(InvocationContext invocationContext) throws Exception{
        return invocationContext.proceed();
    }
}
```

Przykład implementacji interceptora CDI - cd.

Ponad to aby interceptor CDI został wywołany musi być określony w pliku *beans.xml*, jak w przykładzie poniżej.

```
<interceptors>
    <class>com.example.demo.ExampleInterceptor</class>
</interceptors>
```

Można dodać wiele interceptorów, wywoływane są one w takiej samej kolejności jak zostały zapisane do pliku *beans.xml*

Wady i zalety Interceptorów CDI

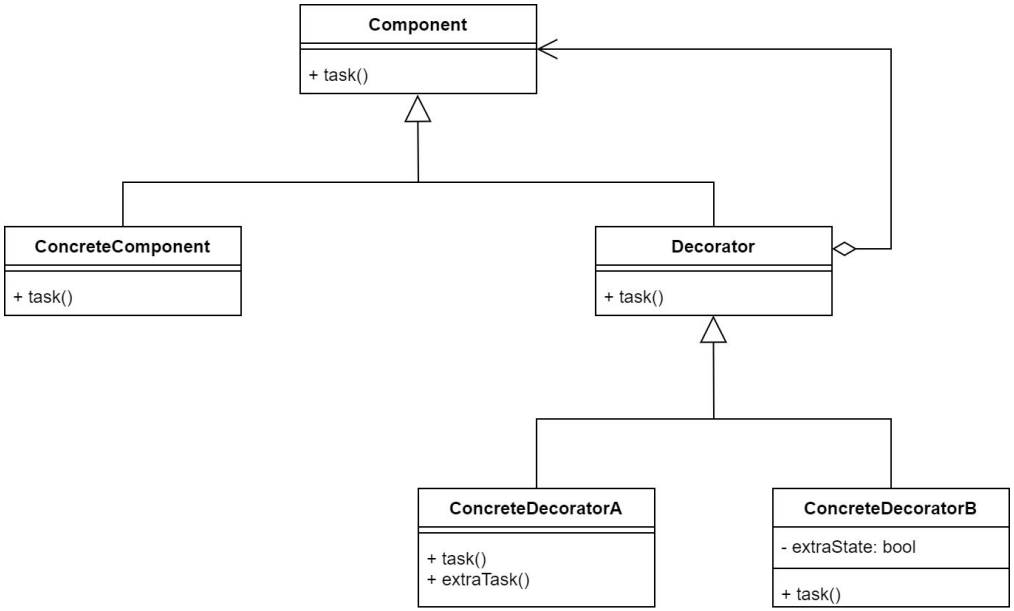
- Zalety
  - główna funkcja Jakarta EE
  - może być używany gdy projekt ma ograniczenia dotyczące bibliotek zewnętrznych
  - niektóre biblioteki CDI mogą być używane w Java SE
- Wady
  - ścisłe połączenie klasy interceptora, a logiki biznesowej
  - brak mechanizmów pozwalającego na zastosowanie interceptorów do grupy metod

Dekorator

Co to za wzorzec?

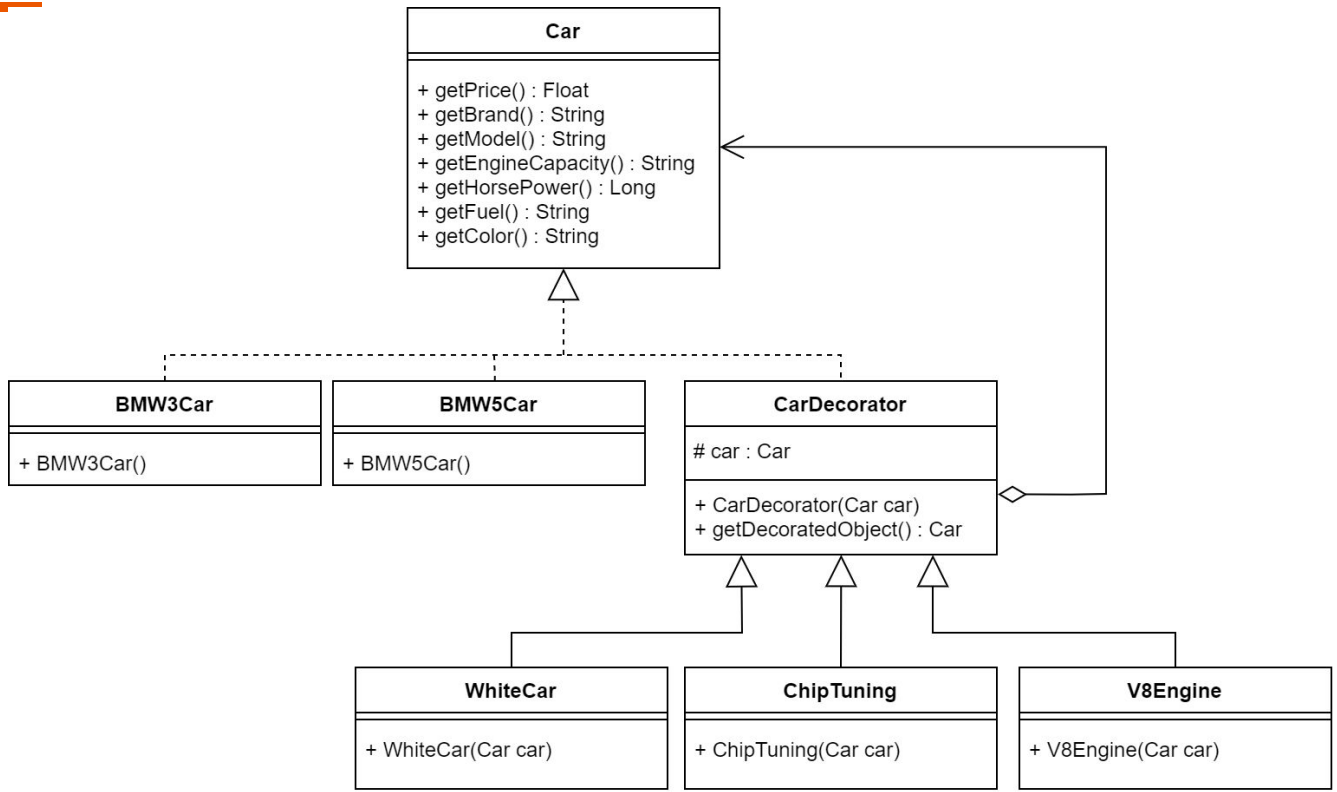
- Należy do grupy wzorców strukturalnych obiektowych, które polegają na składaniu obiektów w celu obsługi nowych funkcji
- Polega on na rekurencyjnym składaniu obiektów, co umożliwia dynamiczne dołączanie dowolnej ilości nowych funkcjonalności
- Umożliwia on większość elastyczność, ponieważ obiekt zmieniany jest w czasie wykonywania programu, a nie podczas jego budowania

Przykładowa implementacja



- **Component** - klasa abstrakcyjna, bądź interfejs
- **ConcreteComponent** - klasa dziedzicząca po klasie Component
- **Decorator** - klasa dziedzicząca po klasie Component oraz zawierająca jej instancje
- **ConcreteDecoratorA/B** - klasy, które dodają nowe metody oraz pola

Przykładowa implementacja - cd.



Przykładowa implementacja - cd.

```java
public interface Car {

    Float getPrice();

    String getBrand();

    String getModel();

    Long getEngineCapacity();

    Long getHorsePower();

    String getFuel();

    Color getColor();

    default String getCarInfo() {
        return String.format(
                " Brand: %s \n Model: %s \n Price: %.2f PLN \n Engine Capacity: %d \n Horse Power: %d \n Fuel: %s \n Color: %s" ,
                getBrand() , getModel() , getPrice() , getEngineCapacity() , getHorsePower() , getFuel() , getColor().name()
        );
    }

}
```

```java
public abstract class CarDecorator implements Car {

    protected Car car;

    protected CarDecorator(Car car) {
        this.car = car;
    }

    public Car getDecoratedObject() {
        return this.car;
    }

}
```

```java
public class BMW5Car implements Car {

    public BMW5Car() {
    }

    @Override
    public Float getPrice() {
        return 200000f;
    }

    @Override
    public String getBrand() {
        return "BMW";
    }

    @Override
    public String getModel() {
        return "5";
    }

    @Override
    public Long getEngineCapacity() {
        return 2500L;
    }

    @Override
    public Long getHorsePower() {
        return 220L;
    }

    @Override
    public String getFuel() {
        return "PB";
    }

    @Override
    public Color getColor() {
        return Color.BLACK;
    }

}
```

```java
public class V8Engine extends CarDecorator {

    public V8Engine(Car car) {
        super(car);
    }

    @Override
    public Float getPrice() {
        return car.getPrice() + 200000f;
    }

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

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

    @Override
    public Long getEngineCapacity() {
        return 5000L;
    }

    @Override
    public Long getHorsePower() {
        return 620L;
    }

    @Override
    public String getFuel() {
        return "PB";
    }

    @Override
    public Color getColor() {
        return car.getColor();
    }

}
```

Przykładowa implementacja - cd.

```
public class Main {

    public static void main(String[] args) {
        //Create basic CAR
        var bmw5 = new BMW5Car();
        //Change color
        var bmw5WithWhiteColor = new WhiteColor(bmw5);
        //Change engine
        var bmw5WithWhiteColorAndV8Engine = new V8Engine(bmw5WithWhiteColor);
        //Chip tuning
        var bmw5WithWhiteColorAndV8EngineAndChipTuning = new ChipTuning(bmw5WithWhiteColorAndV8Engine);

        var BMW3AllDecorators = new ChipTuning(new V8Engine(new WhiteColor(new BMW3Car())));

        System.out.println("RAW Car \n" + bmw5.getCarInfo());
        System.out.println("White color decorator \n" + bmw5WithWhiteColor.getCarInfo());
        System.out.println("White color decorator + V8Engine \n" + bmw5WithWhiteColorAndV8Engine.getCarInfo());
        System.out.println("White color decorator + V8Engine + Chip tuning \n" + bmw5WithWhiteColorAndV8EngineAndChipTuning.getCarInfo());
        System.out.println("BMW3 with all decorators \n" + BMW3AllDecorators.getCarInfo());

    }

}
```

Wzorzec Dekorator - podsumowanie

- Wzorzec projektowy dekorator pozwala na dowolne modyfikowanie istniejących już obiektów bez wpływu na działanie programu
- Możemy powrócić do obiektu przed udekorowaniem dzięki przechowywanej referencji - pozwala to na bezinwazyjne zmiany fragmentów kodu
- Sprawdza się on bardzo dobrze w przypadkach gdzie mamy zaawansowaną hierarchię klas modelowych i zależności między nimi

```
RAW Car
 Brand: BMW
 Model: 5
 Price: 200000.00 PLN
 Engine Capacity: 2500
 Horse Power: 220
 Fuel: PB
 Color: BLACK
White color decorator
 Brand: BMW
 Model: 5
 Price: 205000.00 PLN
 Engine Capacity: 2500
 Horse Power: 220
 Fuel: PB
 Color: WHITE
White color decorator + V8Engine
 Brand: BMW
 Model: 5
 Price: 405000.00 PLN
 Engine Capacity: 5000
 Horse Power: 620
 Fuel: PB
 Color: WHITE
White color decorator + V8Engine + Chip tuning
 Brand: BMW
 Model: 5
 Price: 415000.00 PLN
 Engine Capacity: 5000
 Horse Power: 720
 Fuel: PB
 Color: WHITE
BMW3 with all decorators
 Brand: BMW
 Model: 5
 Price: 365000.00 PLN
 Engine Capacity: 5000
 Horse Power: 720
 Fuel: PB
 Color: WHITE
```

Dziękujemy za uwagę!

Programowanie aspektowe i wzorce projektowe

Autorzy:

Biliński Szymon
Buła Dariusz
Czarnik Michał
Faron Norbert
Telec Mikołaj