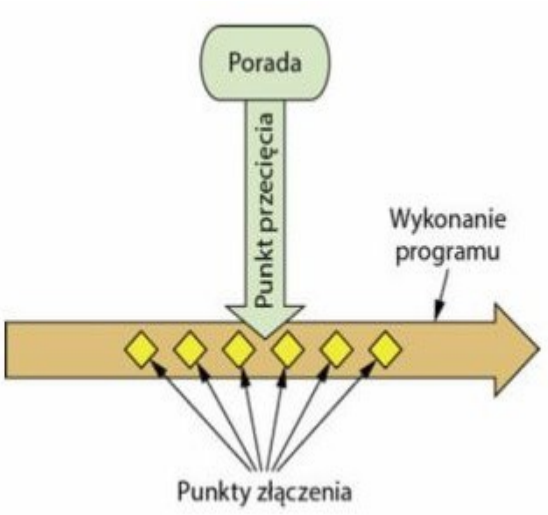
Programowanie komponentowe Spring

Ćw.3 Programowanie aspektowe w Springu

Wstęp

Na zajęciach utworzymy komponenty Spring z dostarczonych klas javy, odpowiednio je skonfigurujemy, utworzymy dla nich pięć rodzajów aspektów a następnie utworzymy dla nich testy jednostkowe.



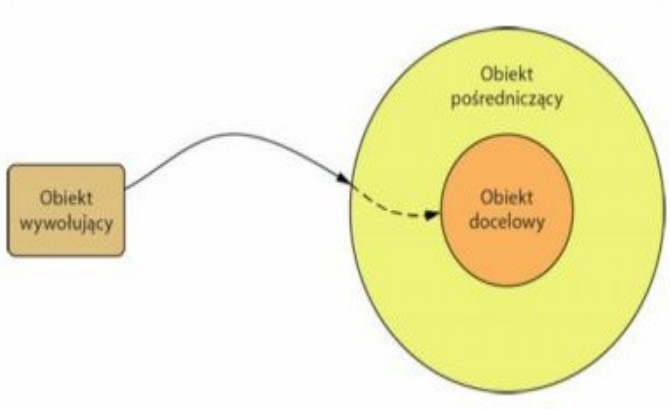
Rys 4.2 Implementacja Aspektów w Springu

Porada = dodatkowa funkcjonalność dołączana do logiki biznesowej

Aspekt = Porada + Punkt przecięcia

Punkt złączenia jest to miejsce w programie, gdzie można ustawić punkt przecięcia aspektu

W AspectJ stosowanym w Springu punkty złączenia są ograniczone do wywołania metod. Nie można ustalić punktu przecięcia ani na zmianę właściwości obiektu ani na wywołanie konstruktora. Powodem tego ograniczenia jest implementacja mechanizmu aspektów za pomocą tworzenia obiektu pośredniczącego, który obudowuje obiekt docelowy. Pozwala to na przejęcie wywołania wszystkich metod i wykonanie przed lub po metodzie dodatkowej funkcjonalnosci, czyli porady.



Rys 4.3 Implementacja aspektów w Springu

Zadanie 1 - Definiowanie aspektów za pomocą konfiguracji Java

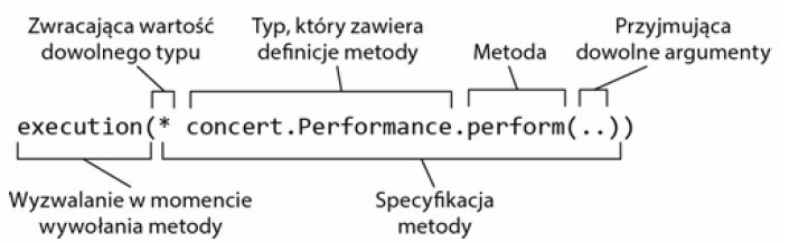
Przypomnienie - podstawowa konfiguracja za pomocą adnotacji Java

- klasa konfiguracyjna powinna być oznaczona adnotacją @**Configuration** i zawierać metody tworzące komponenty oznaczone adnotacją @**Bean**.
- wciśnięcie kombinacji klawiszy Ctrl + Shift + O uruchamia funkcję automatycznego dodawania importów do klasy java
- utworzenie kontekstu

  AnnotationConfigApplicationContext context =
  new AnnotationConfigApplicationContext(
  ***nazwa_klasy_konfiguracyjnej*.class);**
- pobrać z kontekstu referencję do komponentu
  komponent = context.getBean(*nazwa_klasy_komponentu.*class);
- na koniec aplikacji kontekst należy zamknąć przez użycie metody **context.close();**
- wiązanie komponentów przez parametr konstruktora jest realizowane w klasie konfiguracyjnej w metodach oznaczonych adnotacją @**Bean** np.
  return new BraveKnight(quest());

Konfiguracja aspektów za pomocą adnotacji Java

- w celu włączenia mechanizmu aspektów Springa, klasę konfiguracyjną należy dodatkowo oznaczyć adnotacją @**EnableAspectJAutoProxy**
- Klasa zawierająca aspekty powinna być oznaczona adnotacją @**Aspect**



Rys. 4.4 Wyrażenie punktu przecięcia AspectJ

- porada uruchamiana przed wywołaniem metody wybranej klasy jest oznaczana adnotacją **@Before("execution(* *nazwa_pakietu*.*nazwa_klasy*.*nazwa_metody*(..))")**
- porada uruchamiana po poprawnym zakończeniu metody wybranej klasy jest oznaczana adnotacją **@AfterReturning("execution(* *nazwa_pakietu*.*nazwa_klasy*.*nazwa_metody*(..))")**
- porada uruchamiana po rzuceniu wyjątku przez metodę wybranej klasy jest oznaczana adnotacją **@AfterThrowing("execution(* *nazwa_pakietu*.*nazwa_klasy*.*nazwa_metody*(..))")**

Test jednostkowy

- test jednostkowy komponentu Spring wymaga adnotacji:
  @RunWith(SpringJUnit4ClassRunner.class) - oznacza, że klasa używa wsparcia Spring dla testów JUnit
  @ContextConfiguration(classes=*nazwa_klasy_konfiguracyjnej*.class) - określa klasę konfiguracyjną dla kontekstu Spring

Treść zadania

Proszę zaimportować projekt Gradle **concert1**. Projekt zawiera działającą klasę ConcertMain, pustą ConcertConfig, działający test jednostkowy ConcertTest oraz klasę Audience, która zawiera porady bez określenia punktów przecięcia.
Proszę usunąć ścisłe powiązania między klasami i uzupełnić projekt o:

- konfigurację java z komponentami performance i audience
- ConcertMain.java - utworzyć kontekst oparty na konfiguracji java, zamienić bezpośrednie wywołanie konstruktora na pobrać komponent performance z kontekstu, uruchomić koncert, zamknąć kontekst.
- uzupełnić klasę aspektu o odpowiednie adnotacje:
  - dodać aspekt zajmowania przez publiczność miejsc przed koncertem,
  - dodać aspekt wyłączania przez publiczność telefonów przed koncertem,
  - dodać aspekt aplauzu po poprawnym zakończeniu koncertu
  - dodać aspekt żądania zwrotu za bilety po rzuceniu wyjątku podczas koncertu
- uzupełnić testu jednostkowy go o wymagane adnotacje Spring, przetestować komponent i aspekty (bez i z wyrzucaniem wyjątku).

Zadanie 2 - Definiowanie wspólnego punktu przecięcia

Punkt przecięcia

- punkt przecięcia definiujemy przez oznaczenie pustej, bezparametrowej metody adnotacją
  @Pointcut("*wyrażenie_punktu_przecięcia*")
  public *nazwa_metody*() {}
- do tak zdefiniowanego punktu przecięcia odwołujemy się w wyrażeniach przez nazwę metody n.p.
  @Before("*nazwa_metody*()")

Treść zadania

W pierwszym zadaniu wszystkie zdefinowane aspekty miały ten sam punkt przecięcia.
Proszę po zaimportowaniu projektu Gradle **concert2** zdefiniować wspólny punkt przecięcia i użyć go we wszystkich aspektach a następnie utworzyć testy jednostkowe bez i z wyrzucaniem wyjątku.

Zadanie 3 - wykorzystanie porady around

Porada around

- poradę around definiujemy przez użycie adnotacji
  @Around("*wyrażenie_punktu_przecięcia*")
  public *nazwa_metody*(ProceedingJoinPoint jp) {
- parametr **ProceedingJoinPoint jp** umożliwia uruchomienie przechwyconej metody obiektu docelowego przez wywołanie
  jp.proceed();
- przed tym wywołaniem umieszczamy funkcjonalność aspektów @**Before**
- za tym wywołaniem umieszczamy funkcjonalność aspektów @**AfterReturning**
- opakowanie tego wywołania konstrukcją **try{} catch (Throwable e) {}** pozwala na umieszczenie w części **catch** funkcjonalności aspektów @**AfterThrowing**

Treść zadania

Proszę po zaimportowaniu projektu Gradle **concert3** zastąpić wszystkie dotychczasowe aspekty poradą **around** a następnie utworzyć testy jednostkowe bez i z wyrzucaniem wyjątku.

Zadanie 4 - wykorzystanie w aspekcie parametrów metody

Punkt przecięcia z parametrem

- definicję punktu przecięcia z parametrem uzyskamy przez adnotację metody z parametrem
  @Pointcut(("execution(*
  ***nazwa_pakietu.nazwa_klasy.nazwa_metody*(klasa_parametru) &&**
  args(nazwa_parametru)")
  public *nazwa_metody*(klasa_parametru nazwa_parametru) {}

- do tak zdefiniowanego punktu przecięcia odwołujemy się w wyrażeniach przez nazwę metody i parametru n.p.
  @Before("*nazwa_metody*(*nazwa_parametru*)")
  public *nazwa_metody*(*klasa_parametru nazwa_parametru*) {
  funkcjonalność porady - ma pełen dostęp do wartości parametru
  }

Treść zadania

W zadaniu został użyty przykład z pierwszych zajęć, w którym rycerz pokonuje smoka a minstrel to opisuje. Projekt rozszerzono o komponent konia, na którym rycerz wyrusza na misję. Na pierwszych zajęciach aspekty minstrela były zdefiniowane za pomocą pliku xml. W tym projekcie ma zostać wykorzystana konfiguracja oparta na klasie Java z adnotacjami.
Proszę zatem zaimportować projekt Gradle **knight4** i następnie:

- skonfigurować klasę Minstrel jako adnotację za pomocą konfiguracji Java i sprawdzić czy działa,
- skonfigurować poradę @Before Minstrela tak, aby odczytywała wartość parametru horse metody embark i opiewała również konia rycerza (np. nazwa klasy konia),
- uzupełnić testy jednostkowe analogicznie do testów w projektach concert1,2,3.