Moduł 6: Aspekty

Przy okazji omawiania walidacji pojawiły się pewne kwestie, które moim zdaniem wymagają dokładniejszej analizy. Chodzi o sytuację, w której Spring (niezależnie od programisty) uniemożliwiał zapis danych, które nie przeszły walidacji. Co ciekawe, nic podobnego się nie stało, gdy walidacja nie była używana – błędne dane mogły być bez większych przeszkód zapisywane do bazy. Jeżeli przyglądniemy się zwróconemu wyjątkowi, zauważymy że nie pochodzi on z klasy odpowiedzialnej za zapis do bazy danych (a takie mogłoby być pierwsze skojarzenie). Może to nawet lepiej, bo nie burzy to zasady pojedynczej odpowiedzialności w programowaniu obiektowym (ang. single responsibility principle). W naszym konkretnym przypadku zadziałały mechanizmy Hibernate – czyli silnika mapowania obiektowo relacyjnego, który wykorzystywany został do zapisu danych. Niemniej jednak, analogicznego zachowania można oczekiwać korzystając z innych narzędzi realizujących dostęp do bazy danych. Np. korzystając z sql mappera (takiego jak iBatis) aż prosiłoby się o analogiczną funkcjonalność.

W jaki sposób zatem, klasa zupełnie niezwiązana z zapisem danych, została wywołana i zwróciła wyjątek umożliwiający komunikację z bazą danych? Wykorzystana została tutaj funkcjonalność frameworka, czyli możliwość programowania za pomocą aspektów.

Czym są aspekty?

Programowanie z wykorzystaniem aspektów umożliwia dekorowanie klasy dodatkowymi funkcjami, bez ingerencji w samą metodę. Pewnym specyficznym użyciem takiej architektury są filtry w aplikacjach webowych. Aspekty umożliwiają opakowanie wywołania metody dodatkową logiką, która zostanie wywołana przed lub po głównej (obudowywanej) metodzie, jak też może zablokować jej wywołanie w ogóle. Taka architektura jest szczególnie przydatna podczas implementacji wymagań, które wykraczają poza jedną warstwę systemu. Może być to na przykład logowanie, bezpieczeństwo, auditing lub pokazana na wcześniejszych przykładach walidacja. Wykorzystanie aspektów umożliwia modularyzację takich funkcjonalności (które w innym przypadku rozsiane byłby po różnych komponentach całej aplikacji).

Programowanie aspektowe wprowadza szereg nowych pojęć (poza samą nazwą), które warto opisać przed wykorzystaniem tego podejścia:

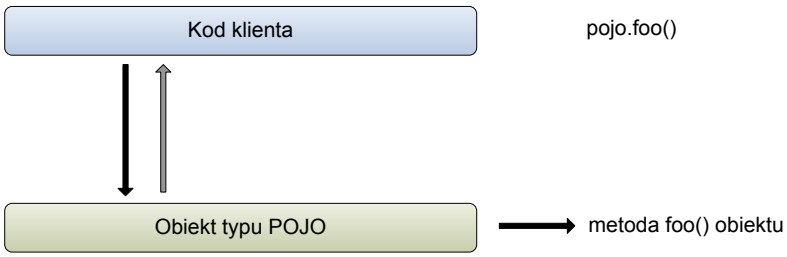
- Advice (porada): akcja, która jest wykonywana w ramach aspektu.
- Join point (czyli punkt złączenia): moment, w którym uruchamiana jest porada, z reguły jest to wywołanie określonej metody.
- Pointcut (punkt przecięcia): predykat, który musi zostać spełniony, aby nastąpiło przecięcie. Język aspektów definiuje szereg zachowań, które można wykorzystać. W najprostszym i najpopularniejszym przypadku, aspekt uzależniony jest od wykonania metody.
- Aspekt: jest to byt łączący wszystkie trzy pojęcia w całość. Aspekt definiuje kod, który zostanie uruchomiany w momencie zaistnienia zdarzenia, określonego w punkcie przecięcia dla metod (lub obiektów) określonych w punkcie złączenia.

Taka definicje mogą wydawać się mało intuicyjne, jednak wszystko stanie się jasne, jeżeli przyglądniemy się przykładom. Nim to nastąpi, musimy jeszcze rozróżnić pięć rodzajów porad:

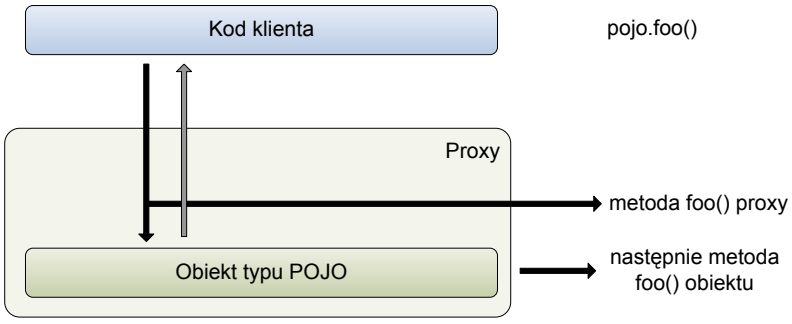
- Before advice: porada uruchamiana jest przed akcją określoną w punkcie złączenia, ale nie ma możliwości jej zablokowania.
- After returning advice: porada uruchamiana jest w momencie poprawnego zakończenia operacji zdefiniowanej w punkcie złączenia (metody nie zwróciła wyjątku).
- After throwing advice: analogicznie do poprzedniego przypadku, z tą różnicą że akcja wywoływana jest w momencie zwrócenia wyjątku przez akcję opisaną w punkcie złączenia.
- After advice: porada łącząca obie poprzednie, czyli wykonywana zawsze po wywołaniu metody opisanej w punkcie złączenia (niezależnie czy wywołanie jest poprawne, czy zwrócony został wyjątek).
- Around advice: najpotężniejszy typ porad, uruchamiany przed wywołaniem akcji określonej w punkcie złączenia, mogący zablokować wywołanie metody lub dodatkowo zinterpretować zwrócony wynik.

Aspekty niskopoziomowe

Rozumienie, do czego i w jaki sposób używać aspektów – to jedno. Omówienie w jaki sposób framework implementuje ten mechanizm – to odmienna kwestia. Jeżeli nie używamy aspektów, klient operuje bezpośrednio na metodach komponentu, co można obrazowo przedstawić na schemacie.



Aspekty działają w sposób przeźroczysty dla użytkownika, bazując na wzorcu projektowym proxy. W momencie, gdy dla któregokolwiek punktu przecięcia zdefiniowana zostanie porada (tzn. kod wywoływany dodatkowo, poza samą logiką komponentu), framework utworzy obiekt proxy, który będzie pośredniczył w komunikacji pomiędzy komponentem a klientem. Zarówno proxy, jak i komponent współdzielą ten sam interfejs – umożliwiają wywołanie takich samych metod, jednakże odpowiedzialnością proxy jest funkcjonalność związana z aspektem, a logika biznesowa delegowana jest do oryginalnego komponentu. Można to obrazowo przedstawić na wykresie.



Praktyczne wykorzystanie aspektów

Aspekty są nierozerwalnie związane ze Springiem, wykorzystywane w bardzo wielu elementach frameworku, często bez wiedzy użytkownika. Można zatem podzielić komponenty na dwie grupy: elementy frameworka umożliwiające deklaratywne wykorzystanie serwisów znanych z platformy JEE

(np. transakcji – które są najistotniejszym serwisem wykorzystującym takie podejście) oraz indywidualnie tworzone przez użytkownika funkcjonalności. W przypadku tych pierwszych rola programisty sprowadza się do deklaratywnej konfiguracji już istniejących komponentów. Jeżeli jednak decydujemy się na pisanie własnych aspektów, pracy jest trochę więcej – i tej grupie skupimy się w niniejszym rozdziale.

Wyobraźmy sobie sytuację, w której w momencie dodania walidatora do kontekstu aplikacji, uruchomione zostają dodatkowe komponenty, których zadaniem jest pilnowanie poprawności przechowywanych danych. Odbywałoby się to poza głównym ciągiem przetwarzania danych, niejako dodatkowo – właśnie do tego celu służą aspekty.

Jak w wielu poprzednich przykładach, skupimy się na konfiguracji poprzez adnotację, aby w dalszej części szkolenia przepisać je przy pomocy konfiguracji XML.

Przygotowanie i konfiguracja porady

Jeżeli decydujemy się na używanie adnotacji `@AspectJ`, to konfiguracja porad wygląda analogicznie do budowania interceptorów na platformie JEE (z tą różnicą, że interceptory zostały ograniczone tylko i wyłącznie do porad Around Advice). Należy także stosować zasady budowy takiej porady: porady nie wymagają implementacji żadnych interfejsów, jedynie zadeklarowania publicznej metody, której argumentem jest obiekt typu `org.aspectj.lang.JoinPoint` (umożliwia do dostęp do kontekstu wywołania i argumentów). Dodatkowo, warto zadbać o to aby metoda interceptora zwracała typ `Object` i deklarowała uniwersalny wyjątek `Throwable`. Wynika to z faktu, że podczas deklarowania interceptora nigdy nie ma pewności, gdzie jego funkcjonalność zostanie wykorzystana. Poprzez zadeklarowanie metody zgodnie z powyższymi zaleceniami, nie należy się obawiać że działania interceptora jakkolwiek złamie kontrakt zadeklarowany przez wywoływaną metodę (a byłoby to niepożądane). Oczywiście, jeżeli będziemy korzystać z innych porad niż Around Invoke – to nie ma konieczności deklarowania wyjątków lub zwracać obiekty o uniwersalnych typach.

Przygotowywany aspekt będzie sprawdzał poprawność danych przekazanych do metody insert sqlMappera, za pomocą zdefiniowanego wcześniej walidatora używanego i zdefiniowanego w aplikacji.

```
@Aspect
@Component
public  class ValidationCheckAdvice {

	@Autowired
	private Validator validator;

	@Around("execution(* com.ibatis.sqlmap.client.SqlMapExecutor.insert(..))")
	public Object validationCheck(ProceedingJoinPoint joinPoint)
		throws Throwable {

		System.out.println("validationCheck() is running!");

		String procedureName = (String) joinPoint.getArgs()[0];
		Object o = joinPoint.getArgs()[1];
		BindException errors = new BindException(o, procedureName);
		validator.validate(o, errors);

		if (errors.hasErrors()) {
			System.out.println(errors.getAllErrors());
			throw new RuntimeException("Cannot insert a not valid object");
		}

		return joinPoint.proceed();
	}

}
```

W powyższym przykładzie wykorzystujemy adnotację AspectJ, zatem konieczne jest poinformowanie o tym frameworka; obywa się to poprzez jeden, dodatkowy wpis w konfiguracji.

```
	<aop:aspectj-autoproxy/>
```

Zacznijmy omawianie przykładu od początku. Adnotacją `@Aspect` informujemy kontener o typie komponentu, a poprzez dodanie adnotacji komponent powodujemy, że aspekt jak automatycznie odnajdowany w procesie skanowania. Bez tej dodatkowej adnotacji, konieczne byłoby jawne dodanie aspektu do kontekstu, w pliku konfiguracyjnym. Nie różniłoby się to w żaden sposób od dodania każdego innego komponentu.

```
	<bean id="validationCheckAvice"
		class="pl.devcastzone.spring.todo.aop.ValidationCheckAdvice"/>
```

Kolejna adnotacja to `@Around`, która definiuje zarówno rodzaj porady, jak i sam punkt złączenia. Możliwe jest rozłączenie tych dwóch operacji (jeżeli pragnęlibyśmy tę samą metodę objąć więcej niż jednym aspektem) poprzez użycie adnotacji `@Pointcut`. W takim przypadku, późniejsze porady będę definiowane w oparciu o nazwę metody oznaczonej adnotacją `@Pointcut`.

```
	@Pointcut("execution(* com.ibatis.sqlmap.client.SqlMapExecutor.insert(..))")
	public void sqlMapperInsert() {}

	@Around("sqlMapperInsert()")
	public Object validationCheck(ProceedingJoinPoint joinPoint)
		throws Throwable {

		System.out.println("validationCheck() is running!");

		String procedureName = (String) joinPoint.getArgs()[0];
		Object o = joinPoint.getArgs()[1];
		BindException errors = new BindException(o, procedureName);
		validator.validate(o, errors);

		if (errors.hasErrors()) {
			System.out.println(errors.getAllErrors());
			throw new RuntimeException("Cannot insert a not valid object");
		}

		return joinPoint.proceed();
	}

	@AfterReturning("sqlMapperInsert()")
	public void afterInsert() {
		System.out.println("Successful insert");
		System.out.println("********");
	}
```

Powyższy przykład ukazuje jeszcze jedną właściwość aspektów: modyfikować można nie tylko wewnętrzne klasy, napisane przez programistę. Aspekty dotyczą wszelkich możliwych klas – tutaj z pakietu iBatis. Dodatkowo, widzimy że aspekt podlega takim samym prawom jak każdy inny komponent Spring i można wstrzykiwać do niego inne komponenty (jak pokazany w przykładzie walidator).

W powyższy sposób, dodatkowo zabezpieczyliśmy naszą aplikację, uniemożliwiając zapis niepoprawnych danych; funkcjonalność typową dla silników ORM (np. Hibernate) udało się uzyskać w sql mapperze iBatis.

Definiowane punktów przecięcia

Dla osób niemających wcześniej kontaktu z programowaniem aspektowym sposób definiowania punktów przecięcia może nie być intuicyjny. Warto zatem poświęcić chwilę na analizę sposobu tworzenia wyrażeń określających punkty złączenia. Wszystkie punkty przecięcia pisane są według tej samej reguły: akcja(typ dla którego akcja jest wykonywana). Typy określane są w typowy dla Javy sposób; jest to albo pełna nazwa klasy (interfejsu) wraz z pakietem lub pełna sygnatura metody. O ile określenie typu jest jednoznaczne (`java.io.Serializable`), to w przypadku metod możliwe jest zastosowanie wieloznaczności (w postaci gwiazdek). Zatem, w pełni określona metoda będzie miała następującą postać: `void pl.devcastzone.spring.service.FooService.action (java.lang.String)`. Z tym że możliwe jest zastosowanie konstrukcji wieloznacznej:

- możliwe jest użycie modyfikatory dostępu (public / protected), a także static itp.
- void pl.devcastzone.spring.service.FooService.action (..) oznaczać będzie wywołanie wszystkich metod – niezależnie od typów argumentu.
- pl.devcastzone.spring.service.FooService.action (..) oznaczać będzie wywołanie wszystkich metod action – niezależnie od typu argumentu oraz zwracanego typu.
- pl.devcastzone.spring.service.FooService.* (..) oznacza wszystkie metody klasy FooService.

- pl.devcastzone.spring.service.*.* (..) oznacza wszystkie metody w pakiecie serwis.
- pl.devcastzone.spring.service..*.* (..) oznacza wszystkie metody w pakiecie serwis oraz w pod pakietach.

Punkty łączenia zdefiniowane tak jak powyżej, używane są do konfiguracji akcji (i w ten sposób zdeklarowany jest komplety punkt przecięcie – ang. *pointcut*). Spring umożliwia użycie następujących akcji `AspectJ` (tych akcji jest znacznie więcej, jednakże użycie którejkolwiek akcji `AspectJ` spoza poniższej listy spowoduje wyjątek `IllegalArgumentException`):

- execution, dla metod – określa wywołanie metody pasującej do szablonu,
- within, dla pakietów – określa wywołanie którejkolwiek punktu łączenia w pakiecie,
- target, dla typów – ogranicza punkty łączenia do obiektów implementujących zadany interfejs (będące w relacji instanceof),
- this, dla typów – podobnie jak w poprzednim przypadku, z tą różnicą, że interfejs musi być implementowany przez obiekt będący proxy,
- args, dla typów – organicza punkty łączenia do metod, gdzie argumenty to klasy wymienionego typu.

Jeżeli akcję within, target lub args poprzedzimy znakiem '@' (@target) – akcje będą odnosiły się do adnotacji.

Powyższe rozważania z oczywistych względów nie wyczerpują tematu, stanowią tylko wprowadzenie do bardzo rozległego tematu. W ogromnej większości przypadków wystarczająca będzie umiejętność rozszerzenia wywołania poszczególnych metod (tak jak zrobione zostało to w przykładzie z walidacją zapisywanych danych), a tworzone aspekty będą na kształt interceptorów znanych ze specyfikacji EJB 3.0.