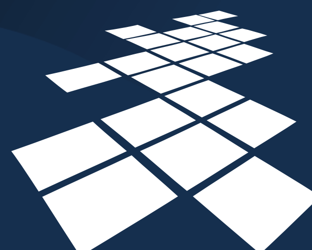
Architektura Spring

wykład prowadzi
Mikołaj Morzy



Plan wykładu

- Wprowadzenie
- Infrastruktura Spring
- Kontener Inversion of Control / Dependency Injection
- Programowanie aspektowe
- Fabryki komponentów
- Spring DAO
- Transakcje w Spring
- Spring MVC

Celem wykładu jest przedstawienie architektury szkieletowej Spring (ang. Spring Framework). Po krótkim wprowadzeniu do architektury Spring zostaną zaprezentowane najważniejsze, z punktu widzenia budowy aplikacji webowych, cechy architektury. W pierwszej kolejności zostanie przedstawiona infrastruktura Spring. Następnie zostaną zaprezentowane dwa najważniejsze mechanizmy stosowane w Springu: mechanizm wstrzykiwania zależności oraz programowanie aspektowe. Kolejna część wykładu zostanie poświęcona koncepcji fabryk komponentów. Spring implementuje wiele wzorców projektowych. Jako przykładowa zostanie przedstawiona implementacja wzorca Data Access Object i szablon do współpracy z interfejsem JDBC. Pokrótce zostaną również omówione transakcje w środowisku aplikacji Spring. Wykład zakończy prezentacja Spring MVC – pełnej implementacji architektury model-widok-kontroler obejmującej wszystkie warstwy aplikacji webowej.

Zaawansowane aplikacje internetowe

Wprowadzenie do architektury Spring

- Kryterium nowoczesności wg. Bittona
  - Jeśli to rozumiesz, to znaczy, że jest już przestarzałe
- Spring Framework
  - Szkielet aplikacyjny Java, Java EE, .NET
  - Bardzo niewielki stopień zależności od Spring API
  - Obejmuje wszystkie warstwy aplikacji
  - Projekt "open source" rozwijany od 2003

Architektura Spring (3)

Architektura Spring jest, niesłusznie, uważana za skomplikowaną, bardzo złożoną i niemożliwą do opanowania. Żartobliwie ujmuje to zdefiniowanie architektury Spring jako nowoczesnej zgodnie z kryterium nowoczesności wg. Bittona: "jeśli to rozumiesz, to znaczy, że jest już przestarzałe". W rzeczywistości architektura Spring stanowi zaawansowany przykład architektury szkieletowej umożliwiającej szybkie rozwijanie dowolnie złożonych aplikacji, niekoniecznie webowych. Oryginalny projekt architektury Spring miał na celu stworzenie środowiska tworzenia aplikacji Java, od graficznego interfejsu użytkownika począwszy, a na aplikacjach konsolowych skończywszy. Szybko jednak okazało się, że architektura Spring może być z powodzeniem wykorzystana na platformie internetowej do tworzenia dużych aplikacji Java EE. Co ciekawe, architektura Spring nie jest ściśle związana z jednym językiem programowania i może być także zastosowana na platformie .NET. Konstrukcja szkieletu architektury Spring powoduje, że jest ona typem tzw. lekkiej infrastruktury (ang. lightweight framework), w której występuje bardzo niewielki stopień zależności od interfejsów Spring API. To ważna cecha infrastruktury oferująca dużą elastyczność. Architektura Spring obejmuje wszystkie warstwy aplikacji i podsuwa rozwiązania, które mogą być stosowane zarówno w warstwie prezentacji, jak i w warstwie integracji i warstwie danych. Architektura Spring jest rozwijana na licencji "open source" od roku 2003 i od tego czasu zdobyła sobie dużą popularność i spore grono osób programujących.

Zaawansowane aplikacje internetowe

Ideologia architektury Spring

- Java EE jest zbyt skomplikowana
- Komponenty encyjne: przestarzałe, złożone, nadużywane i prawie zbędne
- Interfejs czy dziedziczenie?
- Aplikacja zbudowana na komponentach JavaBean
- Za dużo wyjątków w aplikacjach Java
- Niezależne testowanie modułów aplikacji

Architektura Spring (4)

Konstrukcja architektury Spring wynika ze zbioru założeń, które można śmiało nazwać ideologią architektury Spring. Podstawowy zarzut kierowany przez autorów pod adresem technologii Java EE to nadmierne skomplikowanie i zdecydowanie zbyt duża liczba interfejsów i komponentów. Wiąże się z tym konieczność pisania bardzo dużej ilości kodu, który nie wykonuje żadnego przetwarzania, a służy tylko i wyłącznie sklejaniu warstw i komponentów. Największą niechęć autorzy architektury Spring wykazują w stosunku do komponentów encyjnych, które uważają za przestarzałe, zdecydowanie zbyt złożone, używane w sytuacjach, w których proste komponenty JavaBean są zupełnie wystarczające. Podsumowując, autorzy architektury Spring uważają komponenty encyjne za prawie zbędne, czyniąc wyjątek tylko dla aplikacji, których wewnętrzna architektura jest faktycznie rozproszona. Architektura Spring jest oparta praktycznie tylko na interfejsach, mechanizm dziedziczenia jest używany bardzo rzadko. Jest to świadomy wybór projektowy, mający na celu zwiększenie niezależności między aplikacją a Spring API oraz między poszczególnymi warstwami aplikacji. Podstawową cegiełką budulcową aplikacji zgodnych z architekturą Spring są komponenty JavaBean. Jak się okazuje, przy niewielkim wsparciu ze strony kontenera Spring większość złożonej i zaawansowanej funkcjonalności może być osiągnięte przez wykorzystanie współpracujących ze sobą i komunikujących się komponentów JavaBean. Kolejny punkt ideologiczny architektury Spring dotyczy wykorzystania wyjątków. Autorzy wskazują, że większość wyjątków zgłaszanych przez komponenty aplikacji nie ma sensu poza kontekstem komponentu a mechanizm zgłaszania i przechwytywania wyjątków (gdzie w większości przypadków wyjątek jest tylko rejestrowany w dzienniku aplikacji) jest stanowczo nadużywany. Wreszcie, autorzy kładą bardzo silny nacisk na możliwość niezależnego, jednostkowego testowania modułów aplikacji. Część wyborów projektowych całej architektury jest podyktowana koniecznością zapewnienia wygodnego środowiska do testowania modułów w separacji od reszty aplikacji oraz chęcią zapewnienia środowiska do tworzenia aplikacji na podstawie testów, tzw. TDD (ang. test-driven development).

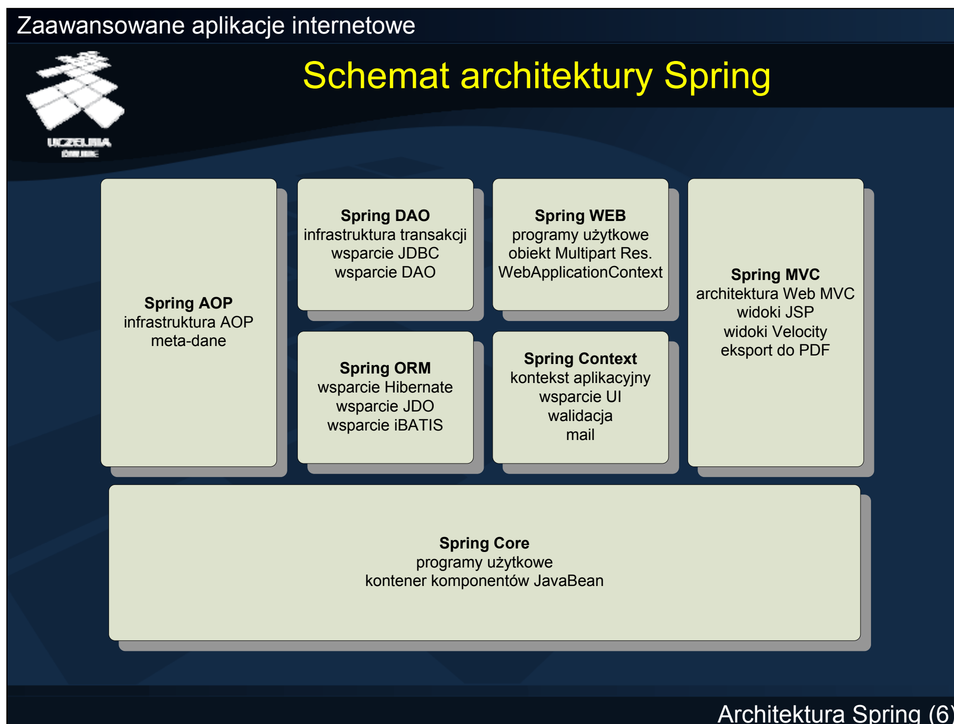
Zaawansowane aplikacje internetowe

Czym jest architektura Spring?

- Wielowarstwowy szkielet aplikacji Java/Java EE/.NET
  - Lekki kontener komponentów JavaBean i POJO
  - Warstwa zarządzania transakcjami
  - Obsługa JDBC wraz z hierarchią wyjątków
  - Integracja z Hibernate, JDO i iBATIS SQL Maps
  - Programowanie aspektowego AOP
  - Model MVC i integracja ze Struts, WebWork, Tapestry
  - Integracja z technologiami szablonów JSP, Velocity Templates, Struts Tiles

Architektura Spring (5)

Architektura Spring to wielowarstwowy szkielet aplikacyjny dla aplikacji pisanych w Javie, Java EE oraz .NET. W skład szkieletu wchodzą: (1) lekki kontener zarządzający komponentami JavaBean i prostymi klasami Java POJO w sposób deklaratywny na podstawie pliku konfiguracyjnego, (2) warstwa zarządzania transakcjami zgodna z wieloma standardami, takimi jak JTA, JDBC, JDO, Hibernate, (3) gotowy szablon obsługujący interfejs JDBC wraz z uproszczoną hierarchią wyjątków zgłaszanych przez źródła danych, (4) moduły w pełni integrujące się z popularnymi narzędziami zarządzania trwałością danych, takimi jak Hibernate, JDO, Toplink, czy iBATIS SQL Maps, (5) kontener umożliwiający pełne wykorzystanie programowania aspektowego, (6) kompletny szkielet aplikacyjny MVC z własnym kontrolerem i kontenerem obsługi JSP i JavaBeans, umożliwiający bezbolesną integrację z istniejącymi szablonami, np. Apache Struts, WebWork czy Tapestry, i wreszcie (7) narzędzia do łatwej integracji z technologiami szkieletowymi warstwy prezentacji, np. Velocity Templates i Struts Tiles. Jak wynika z powyższego slajdu, zakres funkcjonalności architektury Spring jest bardzo szeroki.



Architektura Spring składa się z kilku modułów, z których każdy może być wykorzystywany niezależnie w aplikacji. Oznacza to, że aplikacja może wykorzystywać całą moc architektury Spring, ale może także korzystać z fragmentu architektury, np. z modułu ułatwiającego dostęp do danych. Najważniejszym modułem architektury jest Spring Core, moduł oferujący zaawansowane opcje konfiguracji komponentów JavaBean oraz klas POJO i wykorzystujący technikę wstrzykiwania zależności. Spring AOP to moduł oferujący wsparcie dla techniki programowania aspektowego (ang. aspect-oriented programming). Spring ORM to moduł hermetyzujący usługi zarządzania trwałością i oferujący spójny, prosty i wygodny interfejs do wybranego zbioru takich usług: Hibernate, JDO, Toplink i iBATIS SQL Maps. Z tym modułem ściśle współpracuje moduł Spring DAO zapewniający wsparcie dla dostępu do wielu różnych źródeł danych przez interfejsy JDBC i DAO oraz obsługę transakcji. Moduł Spring Context rozszerza podstawową funkcjonalność modułu Spring Core o internacjonalizację (i18n), obsługę zdarzeń cyklu życia aplikacji, walidację, obsługę poczty elektronicznej, dostęp do interfejsu JNDI, obsługę komponentów EJB, szeregowanie zadań, pracę zdalną czy wreszcie integrację z technologiami szablonów Velocity i FreeMarker. Moduł Spring WEB rozszerza moduł Spring Context o kontekst aplikacji webowej i aspekty specyficzne dla środowiska Web, w tym obsługę ładowania plików na serwer, późne wiązanie parametrów dla obiektów biznesowych, oraz wsparcie dla integracji ze środowiskiem Struts. Moduł Spring MVC to kompletna i w pełni funkcjonalna implementacja architektury MVC wykorzystująca kontener Spring i prowadząca do pełnej separacji logiki sterującej i logiki biznesowej. Moduł Spring MVC jest dosyć podobny do środowiska Struts, choć w dużo mniejszym stopniu związany z technologią JSP (umożliwia generowanie widoków w dowolnej technologii: HTML, Velocity, PDF, Excel). Podstawową zaletą modułu Spring MVC jest wykorzystanie techniki wstrzykiwania zależności w kontenerze sterującym, co ma niebagatelny wpływ na cechy funkcjonalne modułu.

Zaawansowane aplikacje internetowe

Cechy architektury Spring

- Uzupełnienie serwerów aplikacji
- Pełna modułowość szkieletu
- Nie wyważa drzwi otwartych na oścież
  - Log4J, ORM
- Dwa zdania przewodnie
  - "Framework that makes the right thing easy to do"
  - "Finding the simplest thing that can possibly work"

Architektura Spring (7)

Architektura Spring stanowi uzupełnienie funkcjonalności serwerów aplikacji, ale główny nacisk w konstrukcji szkieletu jest położony na aplikację i jej logikę, a nie na konstrukcję serwera. W skutek tego aplikacje pisane zgodnie z architekturą Spring zawierają śladowe ilości kodu integrującego aplikację ze środowiskiem (tzw. "plumbing code"), który nie wykonuje żadnej pożytecznej pracy, ale jest niezbędny w celu uruchomienia aplikacji w danym środowisku. Bardzo istotną cechą Spring jest pełna modułowość – aplikacja może wykorzystać dowolny moduł architektury Spring całkowicie niezależnie od pozostałych modułów. Oczywiście, najlepsze efekty daje konsekwentne wykorzystanie Spring w całej aplikacji, ale nie jest to absolutnie wymagane. Wreszcie, architektura Spring nie wyważa drzwi otwartych na oścież. Jeśli w jakimś aspekcie aplikacji istnieje popularne, sprawdzone i dobre rozwiązanie, jest ono natychmiast integrowane z architekturą. Przykładami mogą być standard zapisu do dzienników aplikacji Log4J oraz techniki odwzorowania obiektowo-relacyjnego. Podsumowując, najlepszą charakterystykę architektury Spring przedstawiają dwa cytaty: "Architektura szkieletowa która powoduje, że pisanie poprawnego kodu jest proste" oraz "Architektura, która stara się znaleźć najprostsze rozwiązanie, które ma szanse zadziałać".

Zaawansowane aplikacje internetowe

Inversion of Control / Dependency Injection

- Zasada Hollywood szukania talentów:
  - "don't call us, we'll call you"!
- Kontener odpowiedzialny za konstrukcję i wiązanie obiektów na podstawie konfiguracji
- Metody implementacji
  - **typ 1**: interfejs do wyszukiwania obiektów zależnych
  - **typ 2**: wstrzykiwanie zależności przez metody setter
  - **typ 3**: wstrzykiwanie zależności przez konstruktor

Architektura Spring (8)

Podstawowym mechanizmem wykorzystywanym w architekturze Spring jest mechanizm wstrzykiwania zależności (ang. dependency injection), zwany także zasadą odwrócenia sterowania (ang. inversion of control). Można ją żartobliwie przedstawić w postaci tzw. "zasady Hollywood szukania talentów" – to nie Pani/Pan dzwoni do nas, to my zadzwonimy do Pani/Pana. W tradycyjnym podejściu aplikacja jest odpowiedzialna za powoływanie do życia nowych obiektów i łączenie obiektów. W przypadku architektury Spring odpowiedzialność za tworzenie i wiązanie (ang. wiring) obiektów spada na kontener IoC, który na podstawie konfiguracji w pliku XML tworzy, w miarę potrzeby, nowe obiekty i łączy je ze sobą oraz ustawia ich cechy. Dzięki temu logika sterująca aplikacji jest całkowicie zwolniona z konieczności tworzenia i wiązania obiektów. Mechanizm wstrzykiwania zależności może być zaimplementowany na trzy sposoby. (1) każdy obiekt może implementować specjalny interfejs wyszukiwania obiektów zależnych, kontener może wołać metody interfejsu na rzecz obiektów, jest to rozwiązanie mało elastyczne i wprowadzające dodatkowy interfejs, a zatem wiążące aplikację z API architektury, (2) każdy obiekt może prezentować swoje cechy na zewnątrz w postaci metod dostępu getter i setter, dla każdej własności kontener może wstrzykiwać zależności przez automatyczne wywoływanie metod setter, (3) kontener może wstrzykiwać zależności na etapie tworzenia obiektu za pomocą publicznego konstruktora. Architektura Spring stosuje zarówno wstrzykiwanie zależności przez metody setter, jak i przez konstruktor.

Inversion of Control - przykład (1/3)

```
public interface MovieFinder { List findAll(); }   (1)

class MovieLister {
  private MovieFinder finder;   (2)

  public MovieLister() { this.finder = new IMDBFinder(); }   (3)

  public Movie[] moviesDirectedBy(String director) {
    List allMovies = finder.findAll();   (4)
    // przetwarzaj liste filmow
  }}
```

trwałe powiązanie

Kolejne slajdy przedstawiają przykład działania kontenera Spring i zasady wstrzykiwania zależności. Pierwszy slajd przedstawia fragment tradycyjnie napisanego kodu. Interesuje nas program potrafiący odczytać bazę danych o filmach i wyświetlić listę filmów wyreżyserowanych przez danego reżysera. W tym celu tworzymy interfejs MovieFinder z metodą findAll() zwracającą kolekcję java.util.List (1). Następnie, tworzymy klasę MovieLister, której składową jest komponent potrafiący odczytywać bazę danych filmów (2). Przyjmijmy, że dysponujemy już klasą IMDBFinder, która implementuje interfejs MovieFinder i potrafi się połączyć z bazą filmów udostępnianych przez serwis www.imdb.com. W tradycyjnym podejściu komponent IMDBFinder zostanie połączony z klasą MovieLister jawnie w kodzie aplikacji, np. w konstruktorze klasy MovieLister (3). Takie rozwiązanie powoduje, że klasa MovieLister staje się trwale powiązana z klasą IMDBFinder i zmiana mechanizmu wyszukiwania filmów (np. na komponent odczytujący listę filmów z lokalnej bazy danych) wymaga ingerencji w kod aplikacji. Wyszukanie filmów wyreżyserowanych przez danego reżysera może polegać na wczytaniu listy wszystkich filmów za pomocą metody findAll() z interfejsu MovieFinder, a następnie przetworzeniu listy wyników i wybraniu filmów danego reżysera (4).

Inversion of Control - przykład (2/3)

```
public interface MovieFinder { List findAll(); }   (1)

class IMDBFinder implements MovieFinder {
  private String url;
  public void setUrl(String url) { this.url = url; }   (2)
  public List findAll() { … }
}

class MovieLister {
  private MovieFinder finder;   (3)
  public void setFinder(MovieFinder finder) { this.finder = finder; }

  public Movie[] moviesDirectedBy(String director) { … }   (4)
}
```

metody *setter*

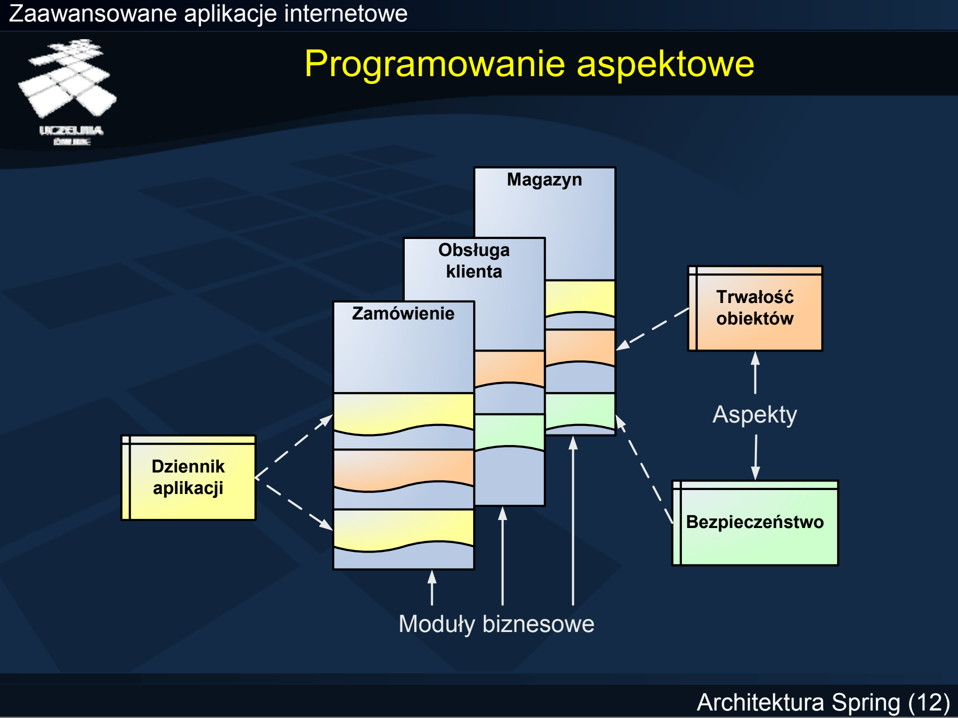
W przypadku wykorzystania mechanizmu wstrzykiwania zależności ten sam kod przyjmuje postać pokazaną na slajdzie. W pierwszej kolejności tworzymy interfejs MovieFinder zawierający metodę findAll() (1). Następnie, tworzymy klasę IMDBFinder implementującą interfejs MovieFinder (2). Istotne jest, że klasa IMDBFinder eksponuje wszystkie swoje składowe za pomocą metod setter. Dalej, budujemy klasę MovieLister zawierającą składową finder, która implementuje interfejs MovieFinder (3). Proszę zauważyć, że w przeciwieństwie do poprzedniego przykładu, w architekturze Spring nie wiążemy jawnie klasy IMDBFinder z klasą MovieLister. Klasa MovieLister również eksponuje wszystkie swoje składowe na zewnątrz w postaci metod setter. Aplikacja polega na kontenerze IoC, który w odpowiedniej chwili, na podstawie pliku konfiguracyjnego, utworzy nowe obiekty i powiąże obiekt klasy IMDBFinder z obiektem klasy MovieLister. Wyszukanie filmów wyreżyserowanych przez danego reżysera odbywa się identycznie jak na poprzednim slajdzie (4). Uwaga: powyższy kod musi być umieszczony w ramach pakietu.

Inversion of Control - przykład (3/3)

beans.xml

```
<beans>
  <bean id="MovieLister" class="spring.MovieLister">
    <property name="finder">              (1)
      <ref local="MovieFinder"/>
    </property>
  </bean>
  <bean id="MovieFinder" class="spring.IMDBFinder">
    <property name="url">
      <value>http://www.imdb.com</value>
    </property>                           (2)
  </bean>
</beans>
```


Slajd przedstawia plik konfiguracyjny beans.xml, który definiuje powiązania między obiektami używanymi w aplikacji. Plik definiuje dwa komponenty, odpowiednio o nazwach MovieLister i MovieFinder. Pierwszy komponent, dla którego implementacją jest klasa spring.MovieLister (której kod został umieszczony na poprzednim slajdzie), posiada składową o nazwie finder (1). Wartością tej składowej jest referencja do innego lokalnego komponentu, a mianowicie referencja do komponentu o nazwie MovieFinder. Komponent ten, którego implementacją jest klasa spring.IMDBFinder, posiada jedną składową o nazwie url i wartości http://www.imdb.com. Powyższy plik konfiguracyjny zostanie wykorzystany przez kontener IoC w następujący sposób. Pierwsze odwołanie do komponentu MovieLister spowoduje utworzenie nowego obiektu klasy spring.MovieLister i wywołanie na rzecz tego obiektu metody setFinder(). Parametrem wywołania metody setFinder() będzie referencja do obiektu klasy IMDBFinder. Jeśli taki obiekt już istnieje, to zostanie wykorzystany, w przeciwnym przypadku kontener utworzy nowy obiekt i wywoła na jego rzecz metodę setUrl() z parametrem "http://www.imdb.com". Referencja do tego obiektu zostanie przekazana jako wartość parametru metody setFinder() w komponencie MovieLister. Oznacza to, że kontener IoC nie tylko automatycznie tworzy i wiąże obiekty, ale również poprawnie określa konieczną kolejność tworzenia obiektów.



Drugą, obok wstrzykiwania zależności, podstawową techniką wykorzystywaną w architekturze Spring jest programowanie aspektowe. W tradycyjnym programowaniu moduły kodu odpowiadają mniej więcej modułom wyodrębnianym na podstawie logiki biznesowej. Przykładowo, w aplikacji przedstawionej na slajdzie wyodrębniono trzy moduły biznesowe odpowiadające obsłudze zamówień, obsłudze klienta i obsłudze magazynu. Równolegle z modułami biznesowymi istnieją także zagadnienia (aspekty) obecne w wielu modułach i związane z architekturą i mechaniką aplikacji a nie logiką biznesową. Przykładami takich zagadnień mogą być: zapewnianie bezpieczeństwa i kontroli dostępu do modułów, zapewnianie trwałości obiektom wykorzystywanym przez moduły, zapisywanie zdarzeń pojawiających się w modułach do dziennika aplikacji, obsługa typowych błędów, itp. Każde z tych zagadnień jest oprogramowywane niezależnie w ramach każdego modułu biznesowego. W rezultacie, kod aplikacji zawiera wymieszane ze sobą fragmenty reprezentujące właściwą logikę biznesową i fragmenty reprezentujące owe zagadnienia przekrojowe. Skutkiem tego jest kod nieczytelny, trudny do modyfikacji i pielęgnacji, podatny na błędy.

Programowanie aspektowe

- Metoda programowania mająca na celu modularyzację kodu i likwidację nakładających się problemów przekrojowych
- Podstawowe pojęcia AOP
  - **aspekt** (aspect): nazwany problem przekrojowy
  - **rada** (advice): dodatkowe zachowanie/czynność które zostaje zainicjowane w punkcie złączenia
  - **punkt złączenia** (join point): miejsce w strukturze wykonywania się aplikacji w którym powinna zostać zastosowana rada

Programowanie aspektowe to metoda programowania polegająca na modularyzacji kodu, identyfikacji problemów przekrojowych, oraz ekstrakcji kodu obsługi problemów przekrojowych w celu likwidacji duplikującego się kodu i wyczyszczenia kodu. Aspekt to nazwany problem przekrojowy występujący w wielu różnych modułach aplikacji. Typowe problemy przekrojowe to: zapisywanie dziennika aplikacji, obsługa błędów, autoryzacja i uwierzytelnianie, zapewnianie trwałości obiektów. Radą nazywamy zachowanie lub czynność, które należy wykonać w określonym miejscu aplikacji. Miejsce zastosowania rady nazywamy punktem złączenia. Zazwyczaj z punktem złączenia związany jest też warunek logiczny określający, czy w danym przypadku należy faktycznie zastosować radę.

Spring AOP

- Rodzaje rad
  - around advice: przed i po punkcie styku, możliwość przerwania przepływu kontroli przez zwrócenie własnej wartości lub wyjątku
  - before advice: przed punktem styku, brak blokowania przepływu kontroli
  - throws advice: w momencie zgłoszenia wyjątku
  - after returning advice: po poprawnym wykonaniu

Moduł Spring AOP udostępnia możliwość definiowania aspektów, rad oraz punktów złączenia. Definiowanie wszystkich elementów ma charakter deklaratywny i odbywa się poprzez pliki konfiguracyjne. Spring AOP umożliwia wykorzystywanie następujących rodzajów rad: (1) around advice: rada wykonuje się przed lub po punkcie złączenia, rada ma możliwość przerwania przepływu kontroli programu przez zwrócenie wartości lub zgłoszenie wyjątku, wykorzystywana do kontroli poprawności wykonania aplikacji i reagowania na sytuacje krytyczne, (2) before advice: rada wykonuje się tylko przed punktem złączenia i nie może przerwać przepływu kontroli, może być wykorzystana np. do transparentnego zapisu do dziennika aplikacji, (3) throws advice: rada wykonuje się w momencie zgłoszenia wyjątku i może posłużyć do transparentnego obsłużenia wyjątków, (4) after returning advice: rada wykonuje się tylko po poprawnym wykonaniu metody, nie może zmienić przepływu kontroli aplikacji i nie może zmienić wartości wynikowej metody, wykorzystywana do obsłużenia poprawnego wykonania się fragmentu kodu. Aktualnie Spring AOP umożliwia wprowadzanie rad tylko w odniesieniu do metod. Doradzanie odbywa się przez przechwytywanie wywołań metod (brak specjalnej kompilacji). Głównym celem jest ścisła integracja Spring AOC i kontenera Spring IoC.

Przykład Spring AOP (1/3)

```
public interface MyInterface { public void foo(); }   (1)

public class MyClass implements MyInterface {
  public void foo() { /* tutaj ciało metody */  }   (2)
}
```

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class TraceBeforeAdvice implements MethodBeforeAdvice {   (3)
  public void before(Method m, Object[] args, Object target)
    throws Throwable { /* tutaj kod rady */ }
}
```


Kolejne trzy slajdy przedstawiają przykład wykorzystania mechanizmu programowania aspektowego w architekturze Spring. Załóżmy, że zdefiniowany został interfejs MyInterface zawierający metodę foo() (1), oraz klasa MyClass implementująca ten interfejs (2). Naszym celem jest stworzenie rady wykonującej się zawsze przed wywołaniem metody foo(). Do tego celu budujemy klasę TraceBeforeAdvice implementującą pochodzący ze Spring API interfejs MethodBeforeAdvice (3). Interfejs ten zawiera metodę before(), do której ciała wpisujemy kod rady. Metoda before() przyjmuje, jako parametry, metodę przed którą ma się wykonać, listę argumentów oraz referencję do obiektu, na rzecz którego wywołuje się obserwowana metoda. Na tym etapie należy podkreślić, że włączenie mechanizmu programowania aspektowego nie wymaga żadnej ingerencji w kod aplikacji (górne okno), a klasa definiująca radę zależy od jednego interfejsu ze Spring API.

Przykład Spring AOP (2/3)

beans.xml

```
<beans>
 <bean id="MyBean"
     class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
     <value>MyInterface</value>
  </property>
  <property name="target"><ref local="beanTarget"/></property>
  <property name="interceptorNames">
   <list><value>theTracingBeforeAdvisor</value></list>
  </property>
 </bean>
 …
```

(1) (2) (3)

Cała konfiguracja programowania aspektowego odbywa się w pliku konfiguracyjnym beans.xml. Śledzeniem wywołań metody foo() zajmie się komponent MyBean zdefiniowany w pliku (1). Komponent ten działa jako obiekt-proxy i śledzi wywołania metod w zadanym obiekcie docelowym. Cecha target określa, jaki inny obiekt należy śledzić (2). Faktycznym śledzeniem zmian zajmie się specjalny komponent, nazwany theTracingBeforeAdvisor (3). Obiekt-proxy i obiekt śledzący zostaną automatycznie powołane do życia przez kontener Spring AOP.

Przykład Spring AOP (3/3)

beans.xml

```
  …
<bean id="beanTarget" class="MyClass"/>                    (1)
<bean id="theTracingBeforeAdvisor"                         (2)
 class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
   <property name="advice">
      <ref local="theTracingBeforeAdvice"/>                (3)
   </property>
   <property name="pattern">                               (4)
      <value>./MyInterface/.foo</value>
   </property>
 </bean>
 <bean id="theTracingBeforeAdvice" class="TracingBeforeAdvice"/>   (5)
</beans>
```


Kolejny komponent zarejestrowany w pliku konfiguracyjnym to obiekt, którego wywołania metody foo() będą śledzone (1). Raz jeszcze należy podkreślić, że aby włączyć śledzenie wywołań metod na rzecz obiektu beanTarget nie jest potrzebna żadna ingerencja w kod tego obiektu. Komponent theTracingBeforeAdvisor (2) to centralny obiekt zajmujący się śledzeniem. Jedną z jego cech jest nazwa rady, theTracingBeforeAdvice (3). Oprócz tego obiekt posiada cechę pattern, określającą wzorzec wywołań metod jakie mają podlegać śledzeniu (4). Innymi słowy, cecha pattern definiuje punkt złączenia dla rady. To wygodne rozwiązanie powoduje, że jeden obiekt śledzący może stosować tę samą radę do wielu miejsc aplikacji. Wzorzec może obejmować wybrane klasy, wybrane metody, może być ograniczony do wybranych pakietów, wreszcie we wzorcu można używać znaków i sekwencji specjalnych (np. można śledzić wszystkie wywołania metod, które posiadają dwa parametry formalne typu String). Ostatni komponent zdefiniowany w pliku konfiguracyjnym to obiekt klasy implementującej radę (5).

Zaawansowane aplikacje internetowe

Spring i AspectJ

- Spring AOP – podstawowa funkcjonalność
- Integracja z AspectJ i AspectWerkz

```
public aspect Tracing {
  public static boolean enabled = false;
  pointcut toBeTraced() : call(*spring.*.*(int,String...)) ||   (1)
           execution(new(..));   (2)

(3) before() : toBeTraced() && if(enabled) {
      Object[] args = thisJoinPoint.getArgs();
      // odczytanie argumentów i zapisanie dziennika
    }
}
```

Architektura Spring (18)

Moduł Spring AOP oferuje podstawową funkcjonalność programowania aspektowego, polegającą na możliwości definiowania aspektów, punktów złączeń, oraz rad. Istnieją bardzo zaawansowane systemy programowania aspektowego, np. AspectJ i AspectWerkz. Spring umożliwia integrację z tymi środowiskami. Slajd przedstawia przykład składni wykorzystywanej w języku AspectJ. Słowo kluczowe pointcut wprowadza definicję punktu złączenia (2), w tym przypadku punktem złączenia jest wywołanie dowolnej metody z klasy umieszczonej w pakiecie zawierającym w nazwie słowo "spring" (wzorzec call *spring.*.*) i posiadającej parametry formalne typu int i String (wzorzec (int,String…)). Drugim alternatywnym warunkiem wystąpienia punktu złączenia jest wywołanie operatora new (wzorzec execute(new(..))). Rada zdefiniowana jest po słowie kluczowym before() (3). Jeśli flaga enabled jest ustawiona na wartość true i prawdziwy jest warunek wystąpienia punktu złączenia toBeTraced(), wówczas wykonuje się kod rady. W pierwszym kroku następuje pobranie argumentów wywołania punktu złączenia, a później może wystąpić dowolny kod obsługi rady.

Fabryki komponentów

- Pakiet **`org.springframework.beans`**
  - **`BeanFactory`** - fabryka komponentów JavaBean
  - **`ApplicationContext`** - obsługa komunikatów, internacjonalizacji, obsługa zdarzeń, rejestrowanie nasłuchu zdarzeń

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
   ClassPathXmlApplicationContext;
class Foo {
  ApplicationContext ctx = new
                ClassPathXmlApplicationContext("beans.xml"); }
```

plik konfiguracyjny

Podstawowa funkcjonalność architektury Spring jest udostępniona przez pakiet org.springframework.beans, który dostarcza infrastrukturę do wyszukiwania i wiązania komponentów JavaBean. Tworzenie i wiązanie komponentów jest wykonywane przez tzw. fabryki komponentów. Spring dostarcza dwóch fabryk komponentów. BeanFactory to podstawowa fabryka komponentów wykorzystująca mechanizm wstrzykiwania zależności do tworzenia i łączenia komponentów JavaBean. ApplicationContext to bardziej złożona i zaawansowana fabryka komponentów, która rozszerza funkcjonalność BeanFactory o umiejętność obsługi komunikatów, internacjonalizacji, obsługi zdarzeń, oraz rejestrowania nasłuchu zdarzeń. Zaleca się, aby poza środowiskami o ograniczonej pamięci (np. applety Java) wykorzystywać fabrykę komponentów ApplicationContext. Na slajdzie przedstawiono fragment aplikacji Java tworzący nową fabrykę komponentów. Fabryka komponentów jest zawsze tworzona w oparciu o podany plik konfiguracyjny.

Komponenty JavaBean

- Brak nadklas bazowych i interfejsów
- Tryby tworzenia komponentów
  - singleton: współdzielony komponent JavaBean
  - prototype: wiele komponentów JavaBean
- Zarządzanie cyklem życia komponentu
  - interfejs **`InitializingBean`**
  - Interfejs **`DisposableBean`**

Komponenty JavaBean stanowią rdzeń architektury Spring. Ciekawe jest to, że architektura Spring nie nakłada na klasy Java stanowiące komponenty JavaBean żadnych dodatkowych wymagań dotyczących rozszerzanych klas lub implementowanych interfejsów. Zakłada się jedynie, że klasa Java (często zwana w tym kontekście POJO, ang. plain old java object) spełnia standardowe wymagania specyfikacji JavaBean, tj. dostarcza domyślny konstruktor, posiada wszystkie składowe prywatne, mieści się w pakiecie i eksponuje swoje składowe za pomocą metod getter i setter. Fabryki komponentów oferują dwa tryby tworzenia komponentów: (1) singleton - fabryka tworzy jedną instancję komponentu JavaBean o podanej nazwie, wszystkie żądania współdzielą jeden i ten sam komponent, (2) prototype: w odpowiedzi na każde żądanie fabryka tworzy nową instancję komponentu JavaBean. Fabryki dostarczają także mechanizmu zarządzania cyklem życia komponentu, interfejsy `InitializingBean` i `DisposableBean` umożliwiają podejmowanie określonych akcji podczas tworzenia i niszczenia komponentu

Przykład (1/4)

Department.java

```
package my.spring;
class Department {
    String name;
    String city;

    public void setName(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void setCity(String city) { this.city = city; }
    public String getCity() { return this.city; }
}
```


Slajd przedstawia klasę komponentu JavaBean reprezentującego departament. Klasa mieści się w pakiecie my.spring i posiada dwie składowe: nazwę departamentu i miasto, w którym mieści się departament. Obie składowe są ukryte przed bezpośrednim dostępem i eksponowane na zewnątrz poprzez odpowiednie metody getter i setter.

Przykład (2/4)

Employee.java

```
package my.spring;
class Employee {
  String name;
  Department dept;

  public void setName(String name) { this.name = name; }
  public String getName() { return this.name; }
  public void setDept(Department dept) {this.dept=dept;}
  public Department getDept() { return this.dept; }
}
```


Slajd przedstawia kod drugiego komponentu JavaBean wykorzystywanego w aplikacji. Podobnie jak poprzednio, klasa Employee spełnia wszystkie wymogi formalne specyfikacji JavaBean. Należy zwrócić uwagę, że jedną ze składowych komponentu jest obiekt klasy Department.

Przykład (3/4)

beans.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="empBean" class="my.spring.Employee">   1
    <property name="name" value="Jan Kowalski"/>
    <property name="dept"><ref bean="deptBean"/></property>
  </bean>                               2

  <bean id="deptBean" class="my.spring.Department" singleton="true">
    <property name="name" value="Internet Applications"/>   3
    <property name="city" value="Poznan"/>
  </bean>                          4
</beans>
```


Slajd przedstawia zawartość pliku konfiguracyjnego beans.xml zawierającego definicje komponentów JavaBean wykorzystywanych w aplikacji. Komponent empBean (1) jest instancją klasy my.spring.Employee, składowa name zostanie wypełniona wartością "Jan Kowalski", natomiast składowa dept zostanie wypełniona referencją do komponentu deptBean (2). Ten komponent jest instancją klasy my.spring.Department. Atrybut singleton="true" (3) oznacza, że do życia zostanie powołana tylko jedna instancja komponentu deptBean i każde odwołanie z aplikacji do tego komponentu będzie się odnosiło do tej właśnie instancji. Podanie wartości dla składowych name i city powoduje, że natychmiast po utworzeniu nowego obiektu my.spring.Department, na rzecz tego obiektu zostaną wywołane metody setName("Internet Applications") i setCity("Poznan").

Przykład (4/4)

SpringTest.java

```
package ploug.spring;
import org.springframework.context.ApplicationContext;                (1)
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringTest {

  public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");   (2)
    Employee employee = (Employee) ctx.getBean("empBean");   (3)

    System.out.println("Name: " + employee.getName());
    System.out.println("Department: " + employee.getDept().getName());
    System.out.println("Location: " + employee.getDept().getLocation());
    ...
```


Slajd przedstawia przykładową aplikację Spring. Jedyne obiekty potrzebne do rozpoczęcia pracy to interfejs fabryki komponentów i konkretna implementacja tego interfejsu. Oba obiekty są w pierwszym kroku importowane (1). Kolejny krok polega na utworzeniu fabryki komponentów na podstawie wskazanego pliku konfiguracyjnego (2). Poza wskazaniem pliku XML na ścieżce zmiennej ClassPath można wskazać plik konfiguracyjny bezpośrednio w systemie plików, wśród zasobów aplikacji webowej, lub poprzez adres URL. Od momentu utworzenia fabryki komponentów przejmuje ona na siebie obowiązek tworzenia, wiązania i zarządzania komponentami. W kroku (3) następuje odczytanie z fabryki komponentu o nazwie empBean. W tym momencie fabryka komponentów wykonuje następujące kroki: sprawdza czy dany komponent istnieje, jeśli nie istnieje, to tworzy ten komponent i wypełnia składowe za pomocą metod setter. W trakcie wstrzykiwania zależności fabryka odkrywa, że konieczne jest utworzenie dodatkowego komponentu o nazwie deptBean (ta zależność jest opisana tylko w pliku konfiguracyjnym, a nie w kodzie aplikacji). Komponent deptBean zostaje automatycznie utworzony i zainicjalizowany, a referencja do tego obiektu zostaje zapisana w komponencie empBean.

Spring DAO

- Wzorzec projektowy Data Access Object (DAO)
  - rozdzielenie mechanizmu trwałości obiektów od reguł biznesowych
- Spring oferuje wsparcie dla
  - interfejsu JDBC
  - narzędzi ORM: Hibernate, JDO, iBATIS SQL Maps
- Dlaczego kolejna warstwa nad JDBC?
  - zarządzanie `Connection`, `ResultSet`, `Statement`
  - uniwersalny wyjątek `DataAccessException`

Data Access Object (DAO) to powszechnie stosowany wzorzec projektowy dostępu do danych zewnętrznych zakładający rozdzielenie kodu odpowiedzialnego za trwałość obiektów od kodu implementującego reguły biznesowe. Spring wspiera konsekwentnie stosowanie tego wzorca i dostarcza rozwiązań ułatwiających implementację wzorca DAO w aplikacjach. Spring oferuje wsparcie zarówno dla interfejsu JDBC, jak i istniejących narzędzi odwzorowania obiektowo-relacyjnego: Hibernate, JDO, iBATIS SQL Maps. Istnieje także możliwość integracji z bardziej złożonymi systemami, np. z Toplink. Według autorów architektury Spring, interfejs JDBC jest w praktyce źródłem bardzo wielu błędów i przyczyną niskiej efektywności aplikacji. W szczególności, ręczne zarządzanie podstawowymi obiektami JDBC, takimi jak Connection, ResultSet, czy Statement prowadzi często do złych rozwiązań. Spring w pełni automatyzuje zarządzanie wszystkimi obiektami JDBC oraz zamienia nieczytelną i skomplikowaną hierarchię wyjątków JDBC na własną uproszczoną i uniwersalną hierarchię wyjątków. Dzięki automatycznemu zarządzaniu obiektami JDBC przestają być konieczne powtarzające się bloki try/catch/finally obsługujące wyjątki konkretnego producenta. Dodatkowo, wszystkie połączenia nawiązywane z bazą danych są zawsze zamykane przez kontener Spring i nie ma niebezpieczeństwa "wycieku" zasobów, w tym przypadku puli dostępnych połączeń. Dostarczona przez Spring API klasa JdbcTemplate umożliwia pisanie w pełni uniwersalnego kodu, który abstrahuje od wykorzystywanej aktualnie bazy danych. Podobnie uniwersalna hierarchia wyjątków automatycznie tłumaczy wyjątki konkretnego dostawcy sterownika JDBC, dzięki czemu warstwa korzystająca ze Spring DAO jest rzeczywiście w pełni niezależna od aktualnej bazy danych.

Zaawansowane aplikacje internetowe

Spring DAO JDBC - przykład (1/2)

SpringDAOTest.java

```
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
...

DriverManagerDataSource dbBean =                                   (1)
                    (DriverManagerDataSource)ctx.getBean("dbBean");
JdbcTemplate template = new JdbcTemplate(dbBean);                  (2)

List result = template.queryForList("SELECT * FROM emp");          (3)
for (Object o : result) {                                          (4)
  Map map = (Map)o;
  System.out.println(map.get("ENAME") + " " + map.get("JOB"));
}
```

Architektura Spring (26)

Slajd przedstawia przykład wykorzystania Spring DAO do odczytania zawartości bazy danych (uwaga: slajd nie jest kompletny, konieczne jest zdefiniowanie pakietu oraz zdefiniowanie klasy i metody main()). W kroku (1) następuje pobranie z kontenera komponentu JavaBean reprezentującego połączenie z bazą danych. Następnie (2), na podstawie istniejącego połączenia tworzony jest obiekt JdbcTemplate. Ten obiekt stanowi hermetyczną reprezentację całej funkcjonalności interfejsu JDBC. Przykład wykonania zapytania to linia (3), wynik zapytania trafia, w postaci kolekcji java.util.List, do obiektu result. Krok (4) to prosta iteracja, w trakcie której dla każdego obiektu znajdującego się w wyniku zapytania zawartość obiektu jest wczytywana do obiektu java.util.Map i wyświetlana na konsoli.

Spring DAO JDBC - przykład (2/2)

beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                  "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  ...
  <bean id="dbBean"
       class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@host:port:sid"/>
    <property name="username" value="student"/>
    <property name="password" value="student"/>
  </bean>
</beans>
```


Slajd przedstawia zawartość pliku konfiguracyjnego beans.xml. Zdefiniowano w nim komponent JavaBean o nazwie dbBean. Komponent ten reprezentuje sterownik JDBC umożliwiający połączenie się z konkretną bazą danych. Taka forma łączenia się z bazą danych jest zalecana tylko dla niewielkich aplikacji lub dla potrzeb testowania. W rzeczywistej aplikacji należałoby wykorzystać interfejs JNDI do zlokalizowania źródła danych. Komponent dbBean posiada jako składowe sterownik JDBC oraz parametry połączenia: adres JDBC-URL, nazwę konta w bazie danych oraz hasło dostępu. Adres JDBC-URL nie jest stały, dla każdego sterownika JDBC adres ma inną postać. Na slajdzie przedstawiono postać preferowaną przez sterownik Oracle JDBC.

Transakcje w Spring

- Wsparcie transakcji w architekturze Spring
  - Interfejs **`PlatformTransactionManager`**
  - Wybór między JTA, Hibernate, JDO, JDBC
  - Szablon **`TransactionTemplate`**
  - Deklaratywne zarządzanie transakcjami za pomocą POJO i programowania aspektowego przy użyciu **`ProxyFactoryBean`** i **`TransactionInterceptor`**,
  - Demarkacja transakcji przez metadane i adnotacje,
  - Jednolity obraz dla transakcji lokalnych i globalnych

Architektura Spring oferuje bogaty wachlarz możliwości przetwarzania transakcyjnego wewnątrz aplikacji. Wybór dotyczy zarówno rodzaju, sposobu demarkacji, oraz sposobu propagacji transakcji. Podstawowym mechanizmem obsługi transakcji jest interfejs PlatformTransactionManager, dla którego istnieje całe mnóstwo implementacji, np. DataSourceTransactionManager, HibernateTransactionManager, JdoTransactionManager, JmsTransactionManager, JtaTransactionManager, PersistenceBrokerTransactionManager, TopLinkTransactionManager, WebLogicJtaTransactionManager. W zależności od wybranej implementacji system zarządzania transakcjami będzie się zachowywał jak JTA, Hibernate, JDO lub JDBC. Alternatywnie, można także wykorzystać gotową klasę pomocniczą TransactionTemplate oferującą metody przetwarzania transakcyjnego. Klasa ta jest bardzo przydatna w przypadku pracy ze źródłami danych, które nie są transakcyjne. Bardzo wygodnym rozwiązaniem jest deklaratywne zarządzanie transakcjami. Przypomina ono rozwiązanie stosowane w EJB, gdzie za pomocą metadanych i adnotacji w pliku konfiguracyjnym można określić granice i cechy poszczególnych transakcji. W przypadku architektury Spring zarządzanie transakcjami może być bardzo efektywnie zrealizowane za pomocą programowania aspektowego. Stąd, Spring API zawiera predefiniowany obiekt-proxy i interceptor transakcji, za pomocą których można łatwo wykorzystać paradygmat programowania aspektowego. Wykorzystanie architektury Spring do zarządzania ma też dodatkowy pozytywny skutek, transakcje lokalne i globalne są widziane dokładnie tak samo.

Transakcje w Spring - przykład (1/2)

beans.xml

```
...
<bean id="dataSource"
   class="org.springframework.jndi.JndiObjectFactoryBean">   (1)
     <property name="jndiName" value="java:comp/env/jdbc/sid"/>
</bean>
<bean id="transactionManager"   (2)
   class="org.springframework.transaction.jta.JtaTransactionManager"/>
<bean id="empManagerJdbcBean"
   class="my.spring.empManagerJdbc">   (3)
    <property name="dataSource"><ref bean="dataSource"/></property>
</bean>
...
```


Slajd przedstawia część pliku konfiguracyjnego beans.xml (przykład kontynuowany na następnym slajdzie). Pierwszy zdefiniowany komponent JavaBean to dataSource (1), komponent ten reprezentuje źródło danych. W przeciwieństwie do poprzednich przykładów, tym razem źródło danych jest poprawnie zdefiniowane w lokalnym drzewie JNDI. Komponent transactionManager (2) to obiekt będący implementacją klasy bibliotecznej JtaTransactionManager. Wreszcie komponent empManagerJdbcBean (3) reprezentuje dane odczytywane z bazy danych. Komponent posiada składową będącą referencją na źródło danych.

Zaawansowane aplikacje internetowe

Transakcje w Spring - przykład (2/2)

beans.xml

```
...
<bean id="empManagerBean"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
   <property name="transactionManager">
         <ref bean="transactionManager"/></property>
   <property name="target"><ref bean="empManagerJdbcBean"/></property>
   <property name="transactionAttributes">
      <props>
         <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
         <prop key="store*">PROPAGATION_REQUIRED</prop>
      </props>
   </property>
</bean>
...
```

1, 2, 3, 4

PROPAGATOIN_REQUIRED, PROPAGATION_SUPPORTS, PROPAGATION_MANDATORY, PROPAGATION_REQUIRES_NEW, PROPAGATION_NOT_SUPPORTED, PROPAGATION_NEVER

Architektura Spring (30)

Slajd przedstawia drugą część pliku konfiguracyjnego beans.xml. Zawiera definicję jednego komponentu JavaBean, obiektu-proxy reprezentującego rzeczywisty obiekt pracownika (1). Jedną ze składowych obiektu-proxy jest referencja do wykorzystywanego zarządcy transakcji (2). Obiekt, dla którego transakcje są zarządzane, jest wskazany jako składowa target (3). Definicja cech, jakie muszą spełniać metody obiektu, są definiowane wewnątrz atrybutu transactionAttributes (4). Przykładowo, definicja na slajdzie mówi, że wszystkie wywołania jakiejkolwiek metody zawierającej w nazwie słowo load* lub store* musi się odbyć w ramach transakcji.

Spring MVC

- Spring MVC przypomina architekturę Struts
  - współdzielony kontroler
- Kontrolery i interceptory wewnątrz kontenera IoC
- Zalety w porównaniu ze Struts:
  - Bardziej elastyczna architektura
  - Serwlety `DispatcherServlet` współdzielą kontekst
  - Łatwa konfiguracja środowiska
  - Brak zależności aplikacji od szkieletu
  - Warstwa widoku niezwiązana z technologią

Architektura Spring zawiera pełną implementację wzorca MVC. W pierwszej chwili implementacja ta bardzo przypomina architekturę Apache Struts. Jednak występują istotne i ważne różnice. W Spring kontroler i interceptory wykonują się wewnątrz kontenera IoC, dzięki czemu konfiguracja środowiska staje się bardzo prosta. Porównując Spring z MVC Struts rzucają się w oczy ulepszenia względem Struts: elastyczna architektura, współdzielenie kontekstu na poziomie aplikacji internetowej, praktycznie zerowa zależność od szkieletu, wreszcie dużo bardziej swobodne powiązanie warstwy prezentacji z konkretną implementacją metody generowania wyniku.

Zaawansowane aplikacje internetowe

Spring MVC

- Zalety w porównaniu ze Struts (ciąg dalszy):
  - Separacja warstw modelu, prezentacji i kontrolera
  - Model niezależny od Spring API lub Servlet API
  - Obiekty modelu do obsługi formularzy
  - Testowanie przy pomocy JUnit
  - Integracja z warstwą pośrednią
  - Integracja z technologiami warstwy prezentacji

Architektura Spring (32)

Spring MVC oferuje bardzo daleko posuniętą separację między warstwami modelu, prezentacji i sterowania. Podobnie jak w innych modułach, tak i w Spring MVC praktycznie nie występuje żadna zależność między aplikacją a Spring API lub Servlet API. Jest to możliwe dlatego, że Spring korzysta z interfejsów a nie z dziedziczenia klas, dzięki czemu całość wiązania obiektów odbywa się na poziomie kontenera. Spring MVC nie wykorzystuje specjalnej klasy do obsługi formularzy, obiekty biznesowe lub transferowe mogą być wykorzystane od zarządzania stanem aplikacji. Konstrukcja Spring MVC stara się umożliwić jak najłatwiejsze testowanie fragmentów aplikacji za pomocą narzędzia testującego, np. JUnit. Możliwość testowania pojedynczych kroków aplikacji zdecydowanie poprawia jakość kodu i przyspiesza pracę grupową nad projektem. Spring oferuje też zaawansowane mechanizmy integracji z warstwą pośrednią, najczęściej niewymagające żadnej ingerencji w oryginalny kod aplikacji. Wreszcie, Spring umożliwia bardzo łatwą integrację aplikacji z istniejącymi technologiami szablonów: Velocity, FreeMarker, WebWork, Tapestry. Aktualnie wciąż jeszcze trwają prace nad integracją Spring Framework i JSF.

Przykład Spring MVC (1/4)

- Zarządca **DispatcherServlet**

web.xml

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app>
 <servlet>
 <servlet-name>empdept</servlet-name>        1
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>        2

 <servlet-mapping>
  <servlet-name>empdept</servlet-name>
  <url-pattern>/empdept/*</url-pattern>
 </servlet-mapping>        3
</web-app>
```


Podstawowym punktem wejścia do aplikacji internetowej jest zarządca. Rolę zarządcy pełni zdefiniowany w pliku konfiguracyjnym web.xml serwlet o nazwie empdept, będący instancją klasy DispatcherServlet (1). Jest on odpowiedzialny za przyjmowanie żądań z zewnątrz i przekierowywanie żądań do właściwych kontrolerów potomnych. DispatcherServlet tworzy kontekst aplikacji internetowej, dostępny z poziomu Spring przez klasę WebApplicationContext. Wymagane jest, aby cała konfiguracja aplikacji internetowej mieściła się w pliku konfiguracyjnym WEB-INF/*servletname*-servlet.xml. Zgodnie z konfiguracją serwlet empdept zostanie załadowany w momencie wystartowania kontenera Spring MVC (2) i do serwletu kontener będzie kierował wszystkie żądania, których adres URL pasuje do wzorca /empdept/* (3)

Przykład Spring MVC (2/4)

empdept-servlet.xml

```
<bean name="/index" class="my.spring.EmpDeptController"/>   (1)
<bean id="dbBean"
  …                                                         (2)
</bean>
<bean
   class="org.springframework.web.servlet.view.InternalResourceViewResolver">
   <property name="prefix"><value>/WEB-INF/jsp/</value></property>
   <property name="suffix"><value>.jsp</value></property>       (3)
   <property name="viewClass">
     <value>org.springframework.web.servlet.view.JstlView</value>
   </property>
</bean>
```


Slajd przedstawia fragment pliku konfiguracyjnego aplikacji internetowej empdept-servlet.xml. Plik definiuje jedno odwzorowanie adresu /index na wywołanie komponentu my.spring.EmpDeptController (1). Oznacza to, że każde żądanie klienta zawierające adres URL /empdept/index zostanie automatycznie przekierowane do wskazanego komponentu. Drugi zdefiniowany komponent, dbBean (2), ma tę samą definicję co komponent na slajdzie 27. Ostatni zdefiniowany komponent (3) jest komponentem pomocniczym, który zajmuje się odwzorowaniem nazw logicznych na fizyczne adresy zasobów. W przykładzie na slajdzie wykorzystany został komponent InternalResourceViewResolver, który każdą logiczną nazwę widoku, np. pracownicy, zamienia na /WEB-INF/jsp/pracownicy.jsp

Zaawansowane aplikacje internetowe

Przykład Spring MVC (3/4)

EmpDeptController.java

```
public class EmpDeptController implements Controller {
  public ModelAndView handleRequest(  (1)
    HttpServletRequest request,HttpServletResponse response)
    throws ServletException, IOException {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");  (2)
    DriverManagerDataSource dbBean =
                              (DriverManagerDataSource)ctx.getBean("dbBean");
    JdbcTemplate template = new JdbcTemplate(dbBean);
    List employeesRaw = template.queryForList("select ename from emp");
    List employees = new ArrayList();
    for (Object o : employeesRaw)
      employees.add(((Map)o).get("ENAME"));
    ModelAndView mv = new ModelAndView("emps");  (3)
    mv.addObject("employees", employees);
    return mv;    } }  (4)
```

Architektura Spring (35)

Slajd przedstawia główny kontroler aplikacji, wywoływany po przesłaniu żądania z adresem URL /empdept/index. Podstawowa metoda obsługi żądań to metoda handleRequest (1). Przyjmuje ona parametry reprezentujące żądanie i odpowiedź, analogicznie do metod doGet() i doPost() w Servlet API. Następnie (2), aplikacja tworzy kontekst i pobiera z niego definicję komponentu dbBean. Ten komponent jest użyty do stworzenia obiektu pomocniczego JdbcTemplate i wydania zapytania do bazy danych. Wynik zapytania zostaje przetransformowany do postaci listy nazwisk, a następnie przekazany do kontekstu aplikacji (4) i zapisany pod nazwą "employees". Od tego momentu wynik zapytania może być pobrany przez dowolny komponent w ramach aplikacji. Wynikiem działania kontrolera jest zwrócony obiekt ModelAndView, zawierający logiczną nazwę widoku emps. Działający w tle komponent InternalResourceViewResolver zamienia tę nazwę na fizyczną lokalizację zasobu: /WEB-INF/jsp/emps.jsp

Przykład Spring MVC (4/4)

/WEB-INF/jsp/emps.jsp

```
<%@ page contentType="text/html;charset=windows-1252" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
  <body>                                    (1)
    <c:forEach var="emp" items="${employees}">
      <c:out value="${emp}"/> <br/>
    </c:forEach>
  </body>
</html>
```


Ostatni slajd przedstawia prostą stronę JSP, która zostanie wywołana przez serwlet sterujący EmpDeptController. Strona korzysta z prostego skryptu do wyświetlenia listy nazwisk wszystkich pracowników. Warto zwrócić uwagę na sposób wykorzystania komponentu employees (1) – został on stworzony i wypełniony wartościami w kontrolerze, a odczytany na stronie JSP.

Materiały dodatkowe

- "Expert one-on-one J2EE Design and Development", R.Johnson, ISBN 0-7645-4385-7, Wrox, 2002
- Spring Framework, http://www.springframework.org
- Spring .NET, http://www.springframework.net