

Spring Framework dla praktyków

Piotr Maj

27 marca 2006

Spis treści

1. Wprowadzenie	1
1.1. Ważne informacje dla czytelników	1
1.2. O autorze	1
1.3. Dla kogo przeznaczona jest ta książka?	2
1.4. Cel	3
1.5. Zakres materiału	3
1.5.1. Omawiane zagadnienia	3
1.5.2. O czym nie jest ta książka?	4
1.6. Rekomendowane lektury	4
1.6.1. Książki	4
1.6.2. Strony WWW	5
1.7. Narzędzia przydatne podczas lektury	5
1.8. Przyjęte konwencje	5
1.9. Podziękowania	5
2. Wprowadzenie do Spring Framework	7
2.1. Kolejny framework?!	8
2.1.1. W mnogości siła?	8
2.1.2. Wady tradycyjnych platform	8

2.1.3. Rozsądny kompromis – Spring Framework	10
2.2. Dlaczego powstał Spring Framework?	11
2.3. Architektura	12
2.4. Zalety	13
3. Bean i kontener IoC	15
3.1. Bean i fabryka beanów	16
3.1.1. Definiowanie prostych beanów	17
3.1.2. Cykl życia	23
3.1.3. Niestandardowe edytory właściwości	25
3.1.4. Bean bez domyślnego konstruktora	27
3.1.5. Tworzenie beanów przy pomocy fabryki	27
3.1.6. Relacje dziedziczenia	30
3.2. Kontener IoC	30
3.2.1. Definiowanie zależności między beanami	31
3.2.2. Sprawdzanie poprawności konfiguracji beanów	34
3.2.3. Automatyczne dopasowywanie zależności	35
4. Programowanie aspektowe w Spring Framework	41
4.1. Wstęp do aspektów	42
4.1.1. Obiekty Proxy	43
4.2. Aspekty pod lupą	45
4.2.1. Koncepcja AOP	45
4.2.2. Sposoby implementacji AOP	47
4.2.3. Zastosowania AOP	49
4.2.4. Frameworki AOP	51
4.3. AOP w Spring Framework	54

4.3.1. org.springframework.aop.framework.ProxyFactoryBean 54

Podsumowanie 55

ROZDZIAŁ 1

Wprowadzenie

wprowadzenie

1.1. Ważne informacje dla czytelników

Niniejsza książka jest dystrybuowana w takiej postaci w jakiej jest.

Autor nie ponosi żadnej odpowiedzialności za ewentualne szkody powstałe w wyniku wykorzystania zawartych w tej książce informacji.

Wszelkie prawa autorskie zastrzeżone. Reprodukacja i redystrybucja w formie oryginalnej lub zmienionej bez pisemnej zgody autora zabroniona.

Kontakt z autorem: pm@jcake.com.

1.2. O autorze

Piotr Maj – absolwent Akademii Ekonomicznej we Wrocławiu (obrona w 2002 r.), programista z zamiłowania, w Javie programuje od 4 lat. Współtwórca i redaktor portalu dla miłośników Javy <http://jdn.pl/>.

Właściciel firmy jcake software <http://jcake.com/> zajmującej się tworzeniem aplikacji internetowych, programów w Javie (server side oraz desktop – Eclipse RCP), a także szkoleniami dot. technologii Eclipse RCP.

Wygłoszone prelekcje:

1. Przenośne aplikacje w Javie – Eclipse RCP – 22.09.2005, wykład na Politechnice Świętokrzyskiej w ramach spotkań Kieleckiej Grupy Użytkowników Linuksa.
2. Przenośne aplikacje w Javie – Eclipse RCP – 26.10.2005, wykład na konferencji Java Techconf organizowanej przez Naukowe Koło Informatyki przy AE w Krakowie oraz serwis jdn.pl.
3. Przenośne aplikacje w Javie – Eclipse RCP – 24.02.2006, wykład dla deweloperów w firmie Rector <http://www.rector.com.pl/>.

1.3. Dla kogo przeznaczona jest ta książka?

Niniejsza książka przeznaczona jest dla wszystkich deweloperów, którzy chcieliby tworzyć nowoczesne aplikacje klasy enterprise w prosty i efektywny sposób. Pokażę, w jaki sposób można ten cel osiągnąć wykorzystując nowoczesną platformę programową – Spring Framework.

Osoby zaczynające dopiero swą przygodę z aplikacjami internetowymi pisаныmi w języku Java, znajdą tu wyczerpujący opis projektu Spring Framework wraz z przykładami jego praktycznego zastosowania.

Ci, którzy wstęp do programowania po stronie serwera mają już za sobą, ale nie mieli dotychczas okazji zapoznać się z projektem Spring Framework książka ta umożliwi poznanie zalet tej pod wieloma względami rewolucyjnej platformy.

Wszyscy użytkownicy Spring Framework znajdą tu także szereg praktycznych porad i gotowych rozwiązań problemów, na które natknął się i z którymi musiał zmierzyć się autor w trakcie wielomiesięcznej pracy z tym projektem.

1.4. Cel

Głównym celem i motywacją do napisania tej książki jest chęć przedstawienia alternatywnego sposobu tworzenia aplikacji internetowych opartego na nieco innych założeniach niż najmodniejsze obecnie podejście wykorzystujące komponenty EJB i serwery aplikacji zgodne ze standardem J2EE. To ostatnie często niepotrzebnie wprowadza duży stopień skomplikowania do programów, sprawia, że są one mało czytelne, charakteryzuje je niska wydajność, a co ważniejsze, często są modelowym przykładem przerostu formy nad treścią.

1.5. Zakres materiału

1.5.1. Omawiane zagadnienia

Zakres tematyczny książki obejmuje wszystko to, co ma do zaoferowania platforma Spring Framework, a więc w szczególności:

- konfigurację kontekstu i kontener IoC,
- wsparcie dla programowania aspektowego w Spring Framework,
- zarządzanie transakcjami,
- omówienie warstwy dostępu do baz danych i utrwalania obiektów,
- zdalne wywoływanie metod w Spring Framework,
- prezentacja implementacji wzorca MVC.

Wszystkie zagadnienia zostały omówione w takim stopniu, aby po przeczytaniu tej książki czytelnik mógł w pełni wykorzystać potencjał Spring Framework we własnym projekcie.

1.5.2. O czym nie jest ta książka?

Wiemy już, jakie zagadnienia zostaną w niniejszej książce poruszone. Teraz słów kilka na temat tego, czym ona nie jest. Przede wszystkim **nie jest to wprowadzenie do świata serwletów, JSP i serwerów aplikacji**. Zagadnienie to samo w sobie jest na tyle ciekawe i złożone, że mogłoby posłużyć za temat do osobnej książki. W tej publikacji zakładam, że czytelnikowi znane są wyżej wymienione zagadnienia w stopniu przynajmniej podstawowym, umożliwiającym samodzielne skonfigurowanie i uruchomienie serwera aplikacji oraz tworzenie deskryptora aplikacji web.xml.

W książce wiele razy pojawią się odwołania do wzorców projektowych, takich jak MVC czy IoC. Niemniej jednak książka ta **nie omawia wzorców projektowych**, a jedynie ich konkretne implementacje. Czytelnik powinien zapoznać się z teorią we własnym zakresie.

1.6. Rekomendowane lektury

1.6.1. Książki

Czytelników, którzy są mniej biegli w programowaniu, zachęcam przed przeczytaniem tej lektury do zapoznania się z publikacjami dotyczącymi bezpośrednio samego języka Java. Do pełnego zrozumienia niniejszej książki ważne jest, aby czytelnik wiedział w szczególności co to są i jak działają mechanizmy odzwierciedleń (ang. *reflection*) oraz dynamicznych pośredników (ang. *dynamic proxy*). Godną polecenia lekturą jest książka *Thinking in Java* autorstwa Bruce'a Eckel'a [?], dostępna nieodpłatnie w formie e-book lub odpłatnie w wersji drukowanej (również w polskiej wersji językowej).

Z zagadnieniami związanymi z serwletami i JSP można się zapoznać dzięki lekturze *Java Servlet* i *Java Server Pages* Marty'iego Hall'a.

1.6.2. Strony WWW

- <http://www.martinfowler.com/> - Strona Martina Fowlera - guru wzorców projektowych

1.7. Narzędzia przydatne podczas lektury

Podczas pisania tej książki autor korzystał wyłącznie z oprogramowania dystrybuowanego na zasadach *open source*. W celu uruchomienia dołączonych przykładów należy zaopatrzyć się w następujące narzędzia:

- Spring Framework (<http://www.springframework.org/>),
- Ant (<http://ant.apache.org/>),
- Tomcat (<http://jakarta.apache.org/tomcat/>),
- hsqldb (<http://hsqldb.sourceforge.net/>),
- Hibernate (<http://www.hibernate.org/>).

Dodakowo potrzeba oczywiście środowiska J2SDK 5.0, które można pobrać ze strony firmy Sun (<http://java.sun.com/j2se/1.5.0/download.jsp>), oraz dowolny edytor kodu lub środowisko IDE (np. Eclipse).

1.8. Przyjęte konwencje

przyjęte konwencje

1.9. Podziękowania

Dziękuję Piotrowi Kobździe za, jak zawsze, cenne uwagi jakimi się ze mną podzielił po przeczytaniu szkicu rozdziału poświęconego teorii AOP.

Dziękuję Michałowi Ciechelskiemu i Tomaszowi Bartkowiczowi za wnikliwe przeczytanie niniejszego opracowania i znalezienie mnóstwa literówek i innych potknięć językowych.

ROZDZIAŁ 2

Wprowadzenie do Spring Framework

Tworzenie złożonych aplikacji nie jest w dzisiejszych czasach rzeczą prostą. Zresztą nigdy taką nie było, ale wszechpanująca obecnie triada *szybciej, lepiej, taniej* znacznie to zadanie komplikuje. Z drugiej strony na sprawę patrząc, w typowych systemach zawsze znajdzie się taka część kodu, która jest co prawda niezbędna do prawidłowego funkcjonowania aplikacji, ale nie jest ściśle związana z jej logiką biznesową, a w dodatku jest powtarzalna i z czasem po prostu nudna do pisania.

Mając na względzie powyższe dwie uwagi możemy szybko dojść do wniosku, że szanse napisania w rozsądnym czasie dobrej i taniej aplikacji (w sensie ilości poświęconych osobogodzin) wzrosną, jeżeli skoncentrujemy się tylko i wyłącznie na faktycznym rozwiązywaniu problemów jej logiki biznesowej, ograniczając do niezbędnego minimum wszystko to, co tejże logiki bezpośrednio nie dotyczy. Logika biznesowa potrzebuje jednak środowiska, w którym będzie mogła swobodnie funkcjonować, a skoro my, gonieni terminami i ograniczeni budżetem, nie mamy czasu na samodzielne jego stworzenie, powinniśmy poszukać gotowego rozwiązania dostarczającego odpowiedniej infrastruktury. W ten sposób dotarliśmy do pojęcia *framework*, czyli aplikacji szkieletowej, która sama w sobie niewiele potrafi, ale za to dostarcza szeregu gotowych narzędzi i wzorców, które umożliwiają deweloperom skoncentrowanie się dokładnie na tym, co w danym projekcie jest najważniejsze.

2.1. Kolejny framework?!

„Od przybytku głowa nie boli” – głosi stare polskie przysłowie, którego autorzy nie mogli zapewne przypuszczać, że powstanie coś takiego jak aplikacja typu framework. . .

2.1.1. W mnogości siła?

Na przestrzeni ostatnich kilku lat powstało co najmniej kilkanaście różnych platform do tworzenia aplikacji internetowych, z których każda realizuje nieco odmienne podejście do tego zagadnienia i oferuje inny zakres funkcjonalny. Część z nich to po prostu implementacja wzorca projektowego MVC (*ang. Model-View-Controller*), której najbardziej znanym przykładem, uważanym *de facto* za standard, jest projekt Struts. Przeciwwagę dla nich stanowią platformy prawie gotowe do użycia i wyposażone w arsenal narzędzi pokrywających swoim zakresem funkcjonalnym cały standard J2EE (np. *Expresso*). Gdzieś pomiędzy plasuje się najwięcej projektów, które implementują wzorzec MVC i jeden lub kilka wybranych aspektów aplikacji internetowych, takich jak zarządzanie dostępem do baz danych czy kontener IoC, etc.

Zorientowanie się w tym wszystkim, jeżeli nawet nie powoduje to bólu głowy to lekkie zawroty napewno. Która platforma jest najlepsza i co skłoniło autorów Spring Framework do stworzenia kolejnej?

2.1.2. Wady tradycyjnych platform

Zanim odpowiemy na powyższe pytanie spróbujmy zastanowić się jakie wady mają dotychczasowe platformy? Oczywiście inne wady można dostrzec w prostych implementacjach wzorca MVC, a inne w kombajnach ze wsparciem dla standardu J2EE.

Głównym argumentem przeciwko tym pierwszym jest to, że **prosty framework jest zbyt prosty, aby mógł znacząco wpłynąć na wzrost wydajności procesu tworzenia aplikacji**, ponieważ:

- wymaga poświęcenia dodatkowego czasu na stworzenie nowych bądź poznanie i integrację gotowych rozwiązań wykonujących brakujące platformie funkcje,
- nie oferuje wsparcia dla transparentnego spełnienia zależności międzykomponentowych.

Framework, który oferuje jedynie podstawowe narzędzia w zakresie obsługi żądań HTTP i kontroli przepływu danych z modelu do widoku to zdecydowanie za mało. Każda aplikacja korzysta z pewnych zasobów zewnętrznych (np. baz danych, serwera SMTP). Jeżeli framework nie oferuje gotowych mechanizmów usprawniających dostęp do zasobów będziemy musieli stracić czas albo na implementację własnych pomysłów, albo na integrację naszego projektu z innymi, komplementarnymi.

Aplikacje składają się przeważnie z wielu komponentów, z których każdy odpowiedzialny jest za pewien wybrany aspekt systemu. Jest wiele przypadków, w których funkcjonalność komponentów w jakimś stopniu się pokrywa. Za prosty przykład niech nam posłuży komponent odpowiedzialny za tworzenie konta użytkownika, który może chcieć wysłać e-mail z potwierdzeniem faktu rejestracji. Aby nie tworzyć dwukrotnie kodu wysyłającego e-mail można utworzyć komponent, którego domeną będzie wysyłanie listów elektronicznych i przekazać referencję tego komponentu do każdego innego, który takiej funkcjonalności potrzebuje. Istnieje wiele sposobów na zrealizowanie tego zadania – niestety proste platformy MVC nie wspierają żadnego z nich, kolejny raz odciągając programistę od rzeczywistej pracy nad systemem. . .

Zaawansowane platformy z kolei pozbawione są przeważnie wyżej wymienionych wad. Niestety, to co ma stanowić o ich sile, a więc kompleksowe podejście do zagadnień infrastruktury często staje się ich największym balastem. Narzędzia te są przeważnie:

- skomplikowane,
- zmuszające do stosowania pewnych, przyjętych przez ich twórców, schematów,

- często nie przystające do specyficznych wymagań naszego projektu.

Stopień złożoności platformy nie pozostaje bez wpływu na wydajność zespołu projektowego, ponieważ trzeba przeznaczyć sporo czasu na jej dokładne poznanie i zrozumienie. Jest to konieczne, jeżeli chcemy w pełni wykorzystać możliwości oferowane przez framework. Decydując się na taki framework tracimy kontrolę nad sposobem implementacji pewnych aspektów, przez co nie tylko platforma, ale i tworzony w oparciu o nią system może być bardziej skomplikowany niż by tego wymagała sytuacja.

Kompleksowe rozwiązania przeważnie narzucają pewne schematy postępowania, które niekoniecznie muszą pokrywać się z naszym rozumieniem dobrych praktyk programistycznych. Niestety, korzystając z gotowca niewiele możemy na to poradzić – musimy zdać się na efekt przemyśleń twórców platformy, rezygnując z własnych. To nie musi być samo w sobie wadą, jednak praktyka pokazuje, że większość programistów i tak tworzy własne alternatywne rozwiązania eliminujące niedostatki platformy, bądź lekko zmieniając jej semantykę.

Celem wielu autorów rozbudowanych platform do tworzenia aplikacji jest stworzenie systemu kompleksowego, który będzie można przystosować do dowolnego typu aplikacji, wreszcie który będzie w stanie zadowolić większość użytkowników. Prowadzi to do pewnych uproszczeń i koncentrowania się na większości typowych przypadków. Niestety, nie zawsze jesteśmy w tej komfortowej sytuacji, że piszemy typowy system, często specyficzne wymagania klienta wykraczają daleko poza te ramy, co wymusza albo przerabianie gotowego rozwiązania albo implementację własnego od podstaw.

2.1.3. Rozsądny kompromis – Spring Framework

Analizując powyższe wady nasuwa się pytanie, jak się ich skutecznie pozbyć nie tracąc jednocześnie niczego z elastyczności i funkcjonalności kompleksowych platform?

Idealem byłby framework, który nie narzuca żadnych gotowych rozwiązań, ale jednocześnie byłby wyposażony we wszelkie mechanizmy ułatwiające

tworzenie aplikacji, z których to mechanizmów moglibyśmy skorzystać niezależnie od potrzeb. Spring Framework stara się realizować to właśnie podejście – dostarczyć budulca bez wskazywania jedynie słusznej drogi jego zagospodarowania.

Choć, jak się później przekonamy, Spring Framework doskonale to zadanie realizuje dystansując pod wieloma względami inne platformy, wcale nie to było głównym celem i motywacją autorów podczas jego tworzenia.

2.2. Dlaczego powstał Spring Framework?

Spring Framework autorstwa Roda Johnsona i Juergena Hoellera jest projektem rozwijanym nieprzerwanie od początku 2003 roku. Sami autorzy wymieniają szereg powodów, które skłoniły ich do jego napisania [?]:

1. **Stworzenie zintegrowanej platformy oferującej solidną infrastrukturę dla podstawowych aspektów każdej aplikacji.** Istnieje wiele ciekawych i godnych uwagi projektów, które obejmują swym zakresem funkcjonalnym pojedynczy aspekt złożonej aplikacji (np. Pico-Container jako kontener IoC, Hibernate jako narzędzie do mapowania O/R). Niestety ich integracja jest czasochłonna i odciąga programistów od zajmowania się rzeczywistymi problemami logiki biznesowej aplikacji. Spring Framework oferuje gotowe do użycia komponenty stanowiące dobrze przetestowaną, stabilną platformę do tworzenia aplikacji.
2. **Minimalizacja stopnia złożoności platformy.** Większość typowych, prostych problemów aplikacji internetowych powinno dać się rozwiązać przy użyciu prostych środków. Zdecydowanie należy unikać zależności od skomplikowanych mechanizmów, jak np. JTA, jeżeli tylko nie ma ku temu wyraźnych powodów.
3. **Nieinwazyjność.** Nasza aplikacja nie powinna, a jeżeli już to tylko w minimalnym stopniu zależeć od infrastruktury platformy. Duży stopień

niezależności można osiągnąć przez zastosowanie kontenera IoC oraz programowania aspektowego.

4. **Prostota testowania.** Pisanie automatycznych testów kodu aplikacji powinno być proste, a poszczególne komponenty systemu łatwo zastępowalne na czas testów obiektami zastępczymi (ang. *mock objects*).
5. **Łatwość rozbudowy.** Platforma powinna promować programowanie zorientowane na interfejsy, a nie na konkretne klasy, co umożliwia proste dostosowywanie jej do własnych potrzeb.

Co ważne, Spring Framework nie jest projektem laboratoryjnym, napisanym w oderwaniu od rzeczywistości. Powstał w procesie ewolucyjnym, a jego przydatność została pozytywnie zweryfikowana przez autorów w kilku dużych komercyjnych projektach, nad którymi mieli okazję pracować.

2.3. Architektura

Spring Framework charakteryzuje się wielowarstwową budową. Poszczególne warstwy są właściwie osobnymi, niezwiązanymi ze sobą podprojektami. Istnieje jednakże jeden element, który umożliwia szybkie i proste zintegrowanie poszczególnych części, a jest nim kontener IoC.

Taki podział czyni ze Spring Framework bardzo elastyczną konstrukcję, którą można dowolnie konfigurować w zależności od potrzeb. Warty podkreślenia jest również fakt, że architektura ta nie ogranicza w żaden sposób Spring Framework do jednego konkretnego zastosowania. Wręcz przeciwnie – framework ten doskonale się sprawdza zarówno w aplikacjach internetowych, jak i okienkowych programach użytkowych.

Na poszczególne warstwy Spring Framework składają się następujące elementy [?]:

- kontener IoC (ang. *Inversion of Control*),
- zarządzanie kontekstem aplikacji,

- wsparcie dla AOP,
- zarządzanie transakcjami,
- abstrakcja DAO,
- wsparcie dla JDBC,
- integracja z narzędziami do mapowania O/R,
- implementacja wzorca MVC,
- wsparcie dla *web-services*.

W kolejnych rozdziałach szczegółowo omówione zostaną poszczególne warstwy Spring Framework.

2.4. Zalety

Czym takim wyróżnia się Spring Framework? Co powinno nas przekonać akurat do tego jednego, spośród kilkunastu dostępnych darmowych frameworków? Moja osobista lista zalet wygląda następująco:

- Spring Framework to rozwiązanie kompleksowe – framework ten oferuje wsparcie dla wszystkich elementów potrzebnych do stworzenia typowej aplikacji,
- Spring Framework implementuje sprawdzone wzorce projektowe i zachęca do ich stosowania. Dzięki temu kod staje się przejrzysty i łatwy do zrozumienia dla każdego, kto te wzorce zna,
- projekt jest doskonale udokumentowany dzięki czemu można bardzo łatwo zrozumieć zasadę jego działania i poznać również zaawansowane funkcje,
- wokół projektu działa silna i prężna społeczność, zawsze gotowa do pomocy (listy mailingowe, forum dyskusyjne).

ROZDZIAŁ 3

Bean i kontener IoC

Głównymi elementami Spring Framework, na bazie których powstały i w ramach których działają wszystkie pozostałe składniki tej platformy są kontener IoC (ang. *Inversion of Control*) i żyjące wewnątrz niego obiekty, zwane beanami.

W rozdziale tym wyjaśnię:

- co to są beany,
- jak je tworzyć i jak z nich korzystać,
- jak definiować zależności międzyobiektove oraz
- jak można wpływać na zachowanie beanów już po ich utworzeniu.

Do pełnego zrozumienia niniejszego rozdziału niezbędne jest posiadanie podstawowej wiedzy na temat następujących zagadnień:

- specyfikacji JavaBeans,
- wzorca projektowego IoC oraz Singleton,
- *lightweight containters*,
- wstrzykiwania zależności przy pomocy konstruktorów i *setterów*.

3.1. Bean i fabryka beanów

Bean w rozumieniu Spring Framework to nic innego jak klasa zgodna ze specyfikacją JavaBeans. Zdaję sobie oczywiście sprawę, że po tak długim wprowadzeniu do tematu czytelnik oczekiwał czegoś bardziej spektakularnego, niemniej jednak taka właśnie jest prawda. Aby zmiejszyć, słuszne poniekąd, rozczarowanie spieszę dodać, że taki stan rzeczy jest jednak bardziej zaletą niż wadą i to co najmniej z kilku powodów.

Najważniejszą zaletą beanów w Spring Framework jest to, że mogą to być absolutnie dowolne obiekty Java. Nie muszą one implementować żadnego interfejsu, ani rozszerzać żadnej specyficznej klasy bazowej – wystarczy, że będą się trzymać znanej z JavaBeans konwencji nazewnicznej metod. Dobrze napisane beany nie zależą więc w żaden sposób od Spring Framework. Raz napisane można ponownie wykorzystać w innym środowisku lub też użyć całkowicie niezależnie jak każdą inną klasę.

Kolejnym ogromnym plusem przyjętego przez autorów Spring Framework podejścia jest to, że bazuje ono na powszechnie przyjętym standardzie. Specyfikacja JavaBeans jest znana praktycznie każdemu programiście, który zetknął się z językiem Java. Odpada więc konieczność uczenia się nowych zagadnień, wejście w świat Spring Framework nie powinno być dla nikogo ani problematyczne ani czasochłonne.

Ponadto, beany, z racji swej prostoty, doskonale ułatwiają praktyczne stosowanie, zyskującego coraz większą popularność, programowania sterowanego testami (ang. *test driven development*). Tworzenie testów do beanów nie trwa długo, podczas ich uruchamiania nie jest wymagana skomplikowana infrastruktura, a większość czasochłonnych, i nie mających bezpośredniego wpływu na logikę beanów operacji da się zasymulować korzystając z obiektów zastępczych, tzw. *mock objects*.

Należy jednak pamiętać, że Spring Framework nie jest w stanie ochronić nas przed błędami projektowymi, jakie możemy popełnić podczas tworzenia własnych beanów. W dalszej części tego rozdziału spróbujemy zlokalizować czyhające na programistę pułapki oraz poszukać sposobów ich uniknięcia.

3.1.1. Definiowanie prostych beanów

Wiemy już, że bean to dowolna klasa zgodna ze specyfikacją JavaBeans. Stworzymy więc nasz pierwszy bean, który zrobi to, co każdy szanujący się pierwszy program powinien zrobić, czyli przywita się ze światem.

Listing 3.1. Pierwszy bean – HelloWorld

```
1 package przyklady.rozdzial3;  
2  
3 public class HelloWorld {  
4     public void hello() {  
5         System.out.println("Witaj świecie!");  
6     }  
7 }
```

Kolejnym krokiem jest poinformowanie Spring Framework o istnieniu naszej klasy. Framework obsługuje standardowo dwa formaty plików konfiguracyjnych. Pierwszy z nich to XML, drugi to standardowy plik **.properties*. Ten ostatni jest niemniej jednak bardzo mało popularny, dlatego też w dalszej części książki wszystkie przykłady będą zapisane wyłącznie w formacie XML.

Dla naszego prostego przykładu plik konfiguracyjny może wyglądać następująco:

Listing 3.2. Minimalny plik konfiguracyjny w wersji XML

```
1 <?xml version="1.0" encoding="UTF-8" ?>  
2 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
3     "http://www.springframework.org/dtd/spring-beans.dtd">  
4 <beans>  
5   <bean id="helloworld" class="przyklady.rozdzial3.HelloWorld" />  
6 </beans>
```

Listing 3.3. Minimalny plik konfiguracyjny w wersji *.properties

```
1 helloworld.class=przyklady.rozdzial3.HelloWorld
```

Na najprostszą definicję beanu składają się: identyfikator oraz pełna kwalifikowana nazwa klasy. Identyfikator umożliwia jednoznaczne odwołanie się do konkretnego beanu.

Nie pozostaje już zatem nic innego, jak tylko zobaczyć nasz bean w akcji. Poniższy kod tworzy dwie identyczne fabryki beanów: pierwszą na podstawie wpisów zawartych w pliku helloworld.xml, a drugą na podstawie pliku

helloworld.properties. Następnie z fabryk pobierany jest nasz przykładowy bean, na którym można już bez przeszkód wywoływać metody biznesowe.

Listing 3.4. Przykład wykorzystania beanu HelloWorld

```
1 package przyklady.rozdzial3;
2
3 import org.springframework.beans.factory.support.DefaultListableBeanFactory;
4 import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
5 import org.springframework.beans.factory.xml.XmlBeanFactory;
6 import org.springframework.core.io.ClassPathResource;
7 import org.springframework.core.io.Resource;
8
9 public class HelloWorldExample {
10
11     public static void main(String [] args) {
12
13         Resource xmlConfigFile =
14             new ClassPathResource("/przyklady/rozdzial3/helloworld.xml");
15         Resource propertiesConfigFile =
16             new ClassPathResource("/przyklady/rozdzial3/helloworld.properties");
17
18         // Tworzenie fabryki na podstawie pliku XML.
19         XmlBeanFactory xmlFactory = new XmlBeanFactory(xmlConfigFile);
20
21         // Tworzenie fabryki na podstawie pliku *.properties.
22         DefaultListableBeanFactory propFactory = new DefaultListableBeanFactory();
23         PropertiesBeanDefinitionReader reader =
24             new PropertiesBeanDefinitionReader(propFactory);
25         reader.loadBeanDefinitions(propertiesConfigFile);
26
27         HelloWorld bean = (HelloWorld) xmlFactory.getBean("helloworld");
28         HelloWorld bean2 = (HelloWorld) propFactory.getBean("helloworld");
29
30         bean.hello();
31         bean2.hello();
32     }
33 }
```

Aby powyższy kod poprawnie się skompilował i uruchomił należy dodać do ścieżki klas archiwum spring.jar oraz pakiety zależne.

Istnieje jeszcze trzeci sposób na utworzenie fabryki beanów, a mianowicie zapisanie jej konfiguracji bezpośrednio w kodzie Java.

Nieco więcej szczegółów na temat beanów

Spring Framework pozwala ściśle określić naturę i moment utworzenia konkretnego beanu. Poniższa definicja jest semantycznie zgodna z tą z listingu 3.2. Odkrywa jednak kilka nowych szczegółów.

Listing 3.5. Bardziej szczegółowy plik konfiguracyjny

```
1 <bean
```

```
2 id="helloworld"  
3 class="przyklady.rozdzial3.HelloWorld"  
4 name="witajswiecie"  
5 singleton="true"  
6 lazy-init="false"/>
```

Pojawiły się trzy nowe atrybuty: `name`, `singleton` oraz `lazy-init`.

- **name** – podobnie jak **id** przypisuje beanowi unikatową nazwę; przy pomocy tego atrybutu można również zdefiniować jeden lub więcej aliasów dla tego samego beana (aliasy oddzielać należy przecinkiem lub średnikiem; spacje są ignorowane),
- **singleton** – określa naturę beanu; jeżeli wartość tego atrybutu została ustawiona na *true* (co jest wartością domyślną) każde pobranie danego beanu z fabryki zwróci zawsze referencję do jego pojedynczej instancji; wartość *false* spowoduje tworzenie za każdym razem nowej instancji beana,
- **lazy-init** – pozwala określić, w którym momencie fabryka powinna utworzyć pojedynczą instancję danego beanu; wartość *false* (domyślna) oznacza, że fabryka powinna utworzyć instancję beanu od razu przy starcie; wartość *true* powoduje odroczenie chwili utworzenia instancji obiektu tak długo, jak to tylko możliwe, czyli do czasu pierwszego pobrania beanu z fabryki.

Szczególną rozważę należy zachować przy stosowaniu atrybutu **singleton**. Domyślnie Spring Framework tworzy pojedyncze, współdzielone instancje każdego beanu i w przeważającej liczbie przypadków jest to działanie pożądane. Taki bean podlega standardowemu cyklowi życia i jest zarządzany przez kontener. Inaczej wygląda sytuacja beanów zdefiniowanych z opcją `singleton=false`. Rola Spring Framework sprowadza się wówczas do tworzenia obiektów i zwracania referencji do nich. Po spełnieniu tego zadania kontener zapomina o ich istnieniu.

Wartość atrybutu `lazy-init` może mieć duży wpływ na czas uruchamiania fabryki. Ma to szczególne znaczenie, gdy zdefiniowane beany korzystają z zasobów zewnętrznych (np. nawiązują połączenia z bazą danych) lub też

wykonywają przy starcie inne czasochłonne czynności (np. obliczeniowe). Opcję leniwej inicjalizacji można aktywować globalnie dla całej fabryki poprzez dodanie atrybutu `default-lazy-init="true"` do elementu `<beans>` w pliku konfiguracyjnym. Wadą takiego podejścia jest to, że Spring Framework nie zasygnalizuje tuż po starcie fabryki ewentualnych błędów w konfiguracji naszych beanów. Jest to więc dobra opcja dla środowiska produkcyjnego – na etapie tworzenia systemu lepiej wyłączyć leniwą inicjalizację beanów, co pozwoli na szybsze wykrywanie ewentualnych błędów w konfiguracji.

Używając atrybutu `lazy-init` możemy również zapobiec tworzeniu przez fabrykę instancji beanów, które nie powinny być tworzone, bo np. służą tylko jako nadrzędne definicje dla innych beanów. Więcej na ten temat w dalszej części tego rozdziału.

Edycja właściwości beanów

Kolejnym ważnym zagadnieniem jest ustalenie stanu początkowego beanu.

Bean'y powinno się pisać tak, aby bez większych, a najlepiej bez żadnych zmian można je było ponownie wykorzystać, np. w innych projektach, dostosowując tylko ustawienia konfiguracji. Konfiguracja nie powinna więc być zaszyta w kodzie klasy, lecz przekazana z zewnątrz. Spring Framework oferuje bardzo prosty, a jednocześnie bardzo elastyczny mechanizm edycji właściwości beanów bazujący na specyfikacji JavaBeans. Wystarczy dodać do beana publiczną metodę ustawiającą pole, czyli tzw. *setter*. Listing 3.6 przedstawia prostą klasę z dwoma publicznymi metodami zgodnymi ze specyfikacją JavaBean.

Listing 3.6. Klasa bez konstruktora domyślnego

```
1 package przyklady.rozdzial3;
2
3 public class SimpleSetter {
4
5     private Integer value;
6     private String name;
7
8     public void setValue(Integer value) {
9         this.value = value;
10    }
11
12    public void setName(String name) {
13        this.name = name;
14    }
15 }
```

```
14 }  
15 }
```

Pola `value` oraz `name` można ustawić w pliku konfiguracyjnym fabryki w następujący sposób:

Listing 3.7. Konfiguracja właściwości beanów

```
1 <bean id="mybean" class="przyklady.rozdzial3.SimpleSetter">  
2   <property name="value"><value>5</value></property>  
3   <property name="name"><value>Hello World!</value></property>  
4 </bean>
```

Spring Framework automatycznie dokona konwersji podstawowych typów znajdujących się w pakiecie `.lang`. Bardziej skomplikowane konwersje (w praktyce dowolne) są również możliwe, ale wymagają napisania rozszerzenia do kontenera. Zagadnieniu temu przyjrzymy się dokładniej w rozdziale 3.1.3.

Uwaga! Zapis `<value></value>` nie oznacza ustawienia wartości pola na `null`. Do tego celu służy specjalny element: `<null/>`.

Mapowanie kolekcji Oprócz edycji typów prostych Spring Framework umożliwi definiowanie w pliku konfiguracyjnym kolekcji obiektów, a w szczególności: list, zbiorów, map oraz szczególnego przypadku mapy – obiektu `java.util.Properties`. Reprezentatywny plik konfiguracyjny mógłby wyglądać następująco:

Listing 3.8. Definiowanie kolekcji

```
1 <bean id="mybean" class="CollectionExample">  
2   <property name="allowed">  
3     <list>  
4       <value>127.0.0.1</value>  
5       <value>192.168.17.100</value>  
6     </list>  
7   </property>  
8   <property name="roles">  
9     <map>  
10      <entry key="bob"><value>admin</value></entry>  
11      <entry key="scott"><value>manager,superuser</value></entry>  
12    </map>  
13  </property>  
14  <property name="countries">  
15    <set>  
16      <value>poland</value>  
17      <value>germany</value>  
18    </set>  
19  </property>  
20  <property name="config">  
21    <props>  
22      <prop key="send.email"><value>>true</value></prop>  
23      <prop key="app.name"><value>MyApplication</value></prop>
```

```
24     </props>
25   </property>
26 </bean>
```

W prawie wszystkich przypadkach kolekcje można dowolnie zagnieżdżać (np. mapa może zawierać listy czy zbiory jako wartości, *etc.*). Wyjątkiem jest tylko element `<props>` odpowiadający obiektowi `java.util.Properties`, który przyjmuje jako wartości tylko łańcuchy tekstowe. Dla podanego przykładu klasa beanu musi zawierać następujące sygnatury metod ¹:

Listing 3.9. Metody zgodne ze specyfikacją JavaBean

```
1 void setAllowed(java.util.List allowed);
2 void setRoles(java.util.Map roles);
3 void setCountries(java.util.Set countries);
4 void setConfig(java.util.Properties configuration);
```

Mapowanie dowolnych beanów jako właściwości innych beanów

Mapowanie typów prostych i kolekcji to za mało, by osiągnąć wszystkie cele. Może się zdarzyć, że zapagniemy utworzyć i przekazać do beanu dowolny obiekt. Na przykład obiekt może zawierać metodę `setConfiguration(Configuration config)`, gdzie `Configuration`, to specjalna klasa, która oprócz przechowywania stanu konfiguracji robi jeszcze kilka innych użytecznych rzeczy. Twórcy Spring Framework przewidzieli taką ewentualność i w prawie każdym miejscu pliku konfiguracyjnego (wyjątek stanowi wspomniany już element `<props>`), w którym wskazuje się wartość można wstawić definicję nowego beana. Ilustruje to listing 3.10.

Listing 3.10. Zagnieżdżony bean

```
1 <bean id="mybean" class="Application">
2   <property name="configuration">
3     <bean class="Configuration">
4       <property name="properties">
5         <props>
6           <prop key="value1"><value>true</value></prop>
7           <prop key="value2"><value>>false</value></prop>
8         </props>
9       </property>
10    </bean>
11  </property>
```

¹Dobrą praktyką jest używanie interfejsów w metodach ustawiających pola zamiast konkretnych klas. Pozwala to na nieinwazyjną zmianę implementacji przekazywanych obiektów.

12 | </bean>

Ostatnią ważną kwestią dotyczącą konfiguracji beanów jest przekazywanie w metodzie ustawiającej referencji do już utworzonego beanu. Zagadnienie zostanie poruszone dalszej części rozdziału, przy okazji omawiania kontenera IoC (rozd. 3.2).

3.1.2. Cykl życia

Tworząc nowe beany fabryka wykonuje pewne czynności w dokładnie zdefiniowanej kolejności. Cykl życia beanu, bo o nim mowa, jest w Spring Framework bardzo prosty i składa się z kilku następujących po sobie etapów:

1. Utworzenie instancji.
2. Ustawienie wartości początkowych pól.
3. Inicjacja.
4. Świadczenie usług przez bean.
5. Zamknięcie beanu.
6. Usunięcie instancji z fabryki.

Uwaga! Powyższy cykl życia dotyczy tylko beanów działających w trybie pojedynczej instancji (`singleton=true`). Jeżeli bean nie jest pojedynczą instancją cykl życia kończy się na etapie inicjalizacji, po przeprowadzeniu której kontener zapomina o istnieniu nowoutworzonego obiektu.

Spring Framework automatycznie kontroluje wszystkie fazy cyklu życia, co w przypadku nieskomplikowanych i mało wymagających beanów jest jak najbardziej pożądane. Niekiedy zachodzi jednak potrzeba przeprowadzenia dodatkowych czynności na etapie inicjalizacji (np. sprawdzenie konfiguracji) czy zamykania beanu (np. zapisanie aktualnego stanu konfiguracji beanu). Właśnie te dwa etapy cyklu życia można zdefiniować samodzielnie. Można to zrobić na dwa sposoby:

- implementując specjalne interfejsy dostarczone przez Spring Framework,
- wskazując w pliku konfiguracyjnym nazwy metod, które mają posłużyć do inicjalizacji i zamknięcia beanu.

Interfejsy wpływające na cykl życia

Spring Framework dostarcza dwa interfejsy, które bean może zaimplementować, aby wpłynąć na swój cykl życia. Są to:

- `org.springframework.beans.factory.InitializingBean` oraz
- `org.springframework.beans.factory.DisposableBean`.

Pierwszy z nich, `InitializingBean` (listing 3.11), definiuje jedną metodę, która zostanie wykonana w momencie pierwszego pobrania beanu z kontenera. Metoda ta może i powinna rzucić wyjątek, jeżeli okaże się, że bean nie jest gotowy do świadczenia usług.

Listing 3.11. Interfejs `InitializingBean`

```
1 void afterPropertiesSet () throws java.lang.Exception;
```

Interfejs `DisposableBean` (listing 3.12) również składa się z pojedynczej metody, którą bean musi zaimplementować. Metoda `destroy()` zostanie wykonana w momencie zamykania fabryki i powinna skutkować zwolnieniem wszystkich zasobów używanych przez bean. Może ona rzucić wyjątek, jeżeli zamknięcie beanu nie jest możliwe. Wyjątek taki zostanie zapisany w dzienniku systemowym, po czym zostanie zignorowany, tj. nie spowoduje nagłego zatrzymania/zniszczenia całej fabryki.

Listing 3.12. Interfejs `DisposableBean`

```
1 void destroy () throws java.lang.Exception;
```

Deklaratywne opisanie cyklu życia

Używanie interfejsów do zasygnalizowania chęci ingerencji w cykl życia beanu ma jedną zasadniczą wadę. Kod beanu staje się całkowicie zależny od Spring Framework. Nie jest możliwe wykorzystanie tak napisanego beanu w innym środowisku bez dokonania stosownych zmian w kodzie. Dla wszystkich osób, które wołałyby zachować niezależność beanów od Spring Framework autorzy tej platformy przewidzieli alternatywny sposób. Można umieścić nazwy metod w pliku konfiguracyjnym, zostaną one wywołane przez Spring Framework za pomocą mechanizmu odbicia (ang. *reflection*).

Listing 3.13. Deklaratywne definiowanie cyklu życia

```
1 <bean
2   id="beanid"
3   class="SampleBean"
4   init-method="initialize"
5   destroy-method="destroy" />
```

Metody określone przez argumenty `init-method` oraz `destroy-method` nie powinny zwracać żadnej wartości, ale mogą deklarować rzucanie dowolnego wyjątku, czyli podlegają dokładnie tym samym regułom, co metody z interfejsów `InitializingBean` oraz `DisposableBean`.

3.1.3. Niestandardowe edytory właściwości

Przedstawione do tej pory możliwości kontenera Spring Framework pozwolą na utworzenie dowolnego beanu. Pora przyjrzeć się bardziej zaawansowanym zagadnieniom, które pozwolą wykorzystać maksimum możliwości tkwiących w Spring Framework.

Pierwszym z nich jest możliwość definiowania własnych edytorów właściwości. Spring Framework umożliwia automatyczną konwersję podstawowych typów, takich jak liczby, czy łańcuchy tekstowe. A co w przypadku np. dat czy innych bardziej złożonych obiektów? Czy można np. automatycznie dokonać konwersji łańcucha tekstowego na datę? Odpowiedź brzmi: tak! Wystarczy napisać własny specjalizowany edytor właściwości i przekazać jego referencję fabryce beanów. Zaraz się okaże, że mimo dość groźnie brzmiącej nazwy nie jest to wcale skomplikowane.

Załóżmy, że chcemy dodać do fabryki możliwość automatycznej konwersji łańcuchów w formacie YYYY-MM-DD na daty. Konfiguracja beana wyglądałaby następująco:

Listing 3.14. Przykład konfiguracji beanu z polem typu `java.util.Date`

```
1 <bean id="mybean" class="przyklady.rozdzial3.CustomEditorSample">
2   <property name="date">
3     <value>2004-10-12</value>
4   </property>
5 </bean>
```

czyli dokładnie tak samo jak każda inna. Bean z kolei posiadałby następującą metodę ustawiającą pole `date`:

Listing 3.15. Metoda ustawiająca pole `date`

```
1 void setDate(java.util.Date date);
```

Spring Framework nie potrafi automatycznie przekonwertować wartości typu `java.lang.String` do typu `java.util.Date`. Należy więc rozszerzyć możliwości fabryki o taką funkcję. Robi się to poprzez zarejestrowanie własnego edytora tuż po utworzeniu fabryki, a przed pobraniem z niej beanów. Poniższy przykład rejestruje jeden edytor, który pozwala edytować pola typu `java.util.Date` o ustalonym formacie.

Listing 3.16. Rejestrowanie własnych edytorów właściwości

```
1 Resource config =
2   new ClassPathResource("przyklady/rozdziel3/customeditor.xml");
3   XmlBeanFactory bf = new XmlBeanFactory(config);
4   bf.registerCustomEditor(java.util.Date.class,
5     new CustomDateEditor(
6       new SimpleDateFormat("yyyy-MM-dd"), true));
7   CustomEditorSample bean = (CustomEditorSample) bf.getBean("mybean");
```

Metoda `registerCustomEditor()` wymaga podania dwóch argumentów:

- typu, który chcemy edytować – musi to być ten sam typ, który przyjmuje metoda ustawiająca pole,
- edytor, implementujący interfejs `java.beans.PropertyEditor`.

Mechanizm rejestrowania edytorów jest więc bardzo elastyczny i umożliwia dokonywanie konwersji dowolnych łańcuchów tekstowych na obiekty i odwrotnie.

3.1.4. Bean bez domyślnego konstruktora

Beanem może być dowolna klasa. W szczególności Spring Framework nie wymaga obecności konstruktora domyślnego. Zamieszczona na listingu 3.17 klasa nie posiada konstruktora domyślnego. Aby Spring Framework mógł utworzyć instancję tej klasy należy w pliku konfiguracyjnym zdefiniować argumenty konstruktora tak jak na listingu 3.18.

Listing 3.17. Klasa bez konstruktora domyślnego

```
1 public class NoDefaultConstructor {
2     public NoDefaultConstructor(String name, Integer count) {
3         this.name = name;
4         this.count = count;
5     }
6 }
```

Listing 3.18. Konfiguracja konstruktora

```
1 <bean id="mybean" class="przyklady.rozdzial3.NoDefaultConstructor">
2     <constructor-arg type="java.lang.String" index="0">
3         <value>Hello World!</value>
4     </constructor-arg>
5     <constructor-arg type="java.lang.Integer" index="1">
6         <value>10</value>
7     </constructor-arg>
8 </bean>
```

Element `<constructor-arg>` może zawierać dwa opcjonalne atrybuty:

- `type` – czyli pełną nazwę typu parametru,
- `index` – czyli nieujemną liczbę określającą, którego parametru definicja dotyczy.

Można pominąć powyższe atrybuty, jeżeli da się jednoznacznie rozpoznać argumenty, czyli w przypadku, gdy są one różnych typów. Element `constructor-arg` może zawierać dowolny element spośród `value`, `list`, `map`, `set`, `properties`, `bean`, `null`, a także referencje do innych beanów.

3.1.5. Tworzenie beanów przy pomocy fabryki

Niekiedy wygodniej jest nie tworzyć beanów bezpośrednio lecz za pośrednictwem fabryk. Może to być również przydatne na przykład w sytuacji, gdy

instancji beanu nie da się z pewnych powodów utworzyć bezpośrednio (np. klasa, która ma być beanem nie jest zgodna ze standardem JavaBeans). Spring Framework oferuje stosowanie fabryk beanów na dwa sposoby:

- poprzez implementację interfejsu `FactoryBean`,
- poprzez użycie atrybutów `factory-bean` oraz `factory-method` w pliku konfiguracyjnym.

Interfejs `org.springframework.beans.factory.FactoryBean`

Bean, który implementuje interfejs `FactoryBean` jest traktowany przez Spring Framework jako szczególny rodzaj beanu. W tym przypadku kontener tworzy instancję beanu, ale nie zwraca referencji do niej lecz zleca jej zadanie utworzenia obiektu docelowego. Interfejs ten definiuje trzy metody:

Listing 3.19. Interfejs `org.springframework.beans.factory.FactoryBean`

```
1 Object getObject() throws Exception;  
2 Class getObjectType();  
3 boolean isSingleton();
```

Metoda `getObject()` zwraca docelowy obiekt utworzony przez fabrykę. Metoda `getObjectType()` zwraca typ docelowego obiektu, ale może też zwrócić wartość `null`, jeżeli fabryka nie jest w stanie zidentyfikować typu przed utworzeniem obiektu docelowego. Ostatnia metoda, `isSingleton()` wskazuje, czy bean ma być jedną współdzieloną instancją, czy też fabryka za każdym razem powinna utworzyć nowy obiekt.

Atrybuty `factory-bean` i `factory-method`

Alternatywę dla interfejsu `FactoryBean` stanowi dopisanie do definicji beanu dodatkowych informacji wskazujących na fabrykę. Poniższy przykład zawiera dwie definicje beanów, które są tworzone przez fabryki.

Listing 3.20. Fabryki beanów

```
1 <!-- fabryka 1 -->  
2 <bean  
3   id="bean1"  
4   class="Bean"
```

```
5     factory-method="createObject">
6
7 <!-- fabryka 2 -->
8 <bean id="factoryBean" class="FactoryBean" />
9 <bean
10     id="bean2"
11     factory-bean="factoryBean"
12     factory-method="createObject" />
```

Klasa `Bean` w fabryce 1 musi posiadać statyczną metodę `createObject()` zwracającą docelowy obiekt. Typ obiektu może być dowolny, w szczególności nie musi to być typ wskazany przez atrybut `class`. Jeżeli jest to ten sam typ to metodę tworzącą można traktować jako alternatywę dla konstruktora. Metoda wskazana w atrybucie `factory-method` może przyjmować dowolną ilość argumentów. Konfiguruje się je za pomocą poznanego już elementu `constructor-arg`, dokładnie w ten sam sposób, jak klasy bez konstruktora domyślnego, z tym wyjątkiem, że nie działa w tym przypadku automatyczne dopasowywanie beanów (zagadnienie to omówione zostanie szerzej przy okazji prezentacji kontenera IoC).

Fabryka 2 działa podobnie, jednak tworzenie instancji beanu nie następuje w tej samej klasie w wyniku wywołania statycznej metody, lecz jest delegowane do innego beanu. Fabryka jest więc w tym przypadku zwykłym beanem, który posiada, już niestaticzną, metodę `createObject`.

Interfejs `FactoryBean` czy atrybuty? Ponieważ Spring Framework oferuje dwa alternatywne mechanizmy stosowania fabryk zasadnym wydaje się pytanie: który stosować i jakich sytuacjach?

Jeżeli możemy sobie pozwolić na zależność od API Spring Framework w naszym projekcie to najwygodniej jest użyć interfejsu `FactoryBean`. Wszystkie rozszerzenia Spring Framework stosują właśnie to podejście. W innym przypadku lepiej zdać się na atrybuty `factory-method` i `factory-bean`.

3.1.6. Relacje dziedziczenia

3.2. Kontener IoC

Z lektury poprzedniego rozdziału wiemy już sporo o beanach, ich powstawaniu, naturze i cyklu życia. Kolejnym fundamentalnym elementem platformy Spring Framework jest kontener IoC umożliwiający definiowanie wzajemnych relacji między beanami. O relacji czy zależności między obiektami (beanami) można mówić wtedy, gdy jeden bean wymaga do prawidłowego funkcjonowania innego obiektu.

IoC, czyli *Inversion of Control* to wzorzec projektowy umożliwiający realizację zależności między obiektami niejako bez wiedzy o tym fakcie samych zainteresowanych. IoC realizuje się najczęściej poprzez wstrzykiwanie zależności (ang. *dependency injection*) umieszczając uprzednio obiekty w specjalnym kontenerze. Kontener sam potrafi poprawnie dopasować wszystkie zależności i skonfigurować obiekty przed udostępnieniem ich użytkownikowi. Więcej na temat wzorca IoC można poczytać w moim artykule „Wprowadzenie do lightweight containers” opublikowanym w portalu JDN (<http://jdn.pl/node/1>).

Spring Framework zawiera bardzo wygodną w użyciu implementację kontenera IoC. Springowy kontener IoC jest sercem całej platformy, wszystkie pozostałe elementy z niego korzystają.

Co ważne kontener IoC w Spring Framework jest również dostępny w postaci samodzielnej biblioteki, którą można zintegrować z własnym projektem bez konieczności korzystania z całego frameworka. W takim przypadku Spring Beans (w postaci archiwum jar o rozmiarze nieco większym niż 200Kb) stanowi doskonałą alternatywę dla innych dostępnych na rynku kontenerów IoC (Picocontainer, HiveMind).

Kontener IoC w Spring Framework jest nazywany również fabryką beanów. Te dwie nazwy stosowane będą w niniejszej książce wymiennie.

3.2.1. Definiowanie zależności między beanami

Istnieje kilka metod spełniania zależności między obiektami, takich jak wyszukiwanie obiektów (ang. *lookup*), wstrzykiwanie zależności przy pomocy konstruktorów, setterów czy pól. Kontener IoC w Spring Framework implementuje dwa sposoby wstrzykiwania zależności:

- za pomocą *setterów* (ang. *setter injection*),
- za pomocą *konstruktorów* (ang. *constructor injection*).

Każde z tych podejść ma swoje wady i zalety, każde ma swoich zwolenników i przeciwników. Nie chcąc wzbudzać świętych wojen poprzestańmy na ustaleniu, że wszędzie stosować będziemy takie podejście, które będzie dla nas w danej chwili bardziej intuicyjne i po prostu wygodniejsze.

Przyjrzyjmy się jak wygląda w Spring Framework wstrzykiwanie zależności. Niech za przykład posłużą nam dwa proste beany `NameProvider` oraz `NameWriter` o następującej postaci:

Listing 3.21. Przykładowe beany - `NameProvider`

```
1 package przyklady.rozdzial3;
2
3 public class NameProvider {
4     public String provideName() {
5         return "Janek";
6     }
7 }
```

Listing 3.22. Przykładowe beany - `NameWriter`

```
1 package przyklady.rozdzial3;
2
3 public class NameWriter {
4
5     private NameProvider nameProvider;
6
7     public NameWriter() {}
8
9     public NameWriter(NameProvider nameProvider) {
10        setNameProvider(nameProvider);
11    }
12
13    public void setNameProvider(NameProvider nameProvider) {
14        this.nameProvider = nameProvider;
15    }
16
17    public void writeName() {
18        System.out.println(nameProvider.provideName());
19    }
20 }
```

```
19     }  
20 }
```

Widzimy więc, że klasa `NameWriter` korzysta z klasy `NameProvider` w celu pobrania imienia. Jak zapisać tę zależność w Spring Framework?

Wstrzykiwanie zależności za pomocą *setterów*

Pierwszym omówionym sposobem wstrzykiwania zależności jest wykorzystanie setterów, czyli metod ustawiających dane pole. W tym przypadku konfiguracja beanów powinna wyglądać następująco (dla czytelności pominięto deklarację DTD):

Listing 3.23. Wstrzykiwanie zależności przy pomocy setterów

```
1 <beans>  
2   <bean id="nameProvider" class="przyklady.rozdzial3.NameProvider" />  
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter">  
4     <property name="nameProvider"><ref bean="nameProvider" /></property>  
5   </bean>  
6 </beans>
```

W linii 4 używając elementu `<ref bean="nameProvider" />` informujemy kontener, że w trakcie tworzenia beana o nazwie `nameWriter` powinien przekazać do settera o nazwie `nameProvider` (czyli konkretnie do metody `setNameProvider()`) beana o nazwie `nameProvider`. Jeśli bean `nameProvider` nie został wcześniej utworzony kontener utworzy go i zainicjuje automatycznie.

Można sprawdzić działanie wstrzykiwania zależności przez settery uruchamiając przykład `przyklady.rozdzial3.SetterInjectionExample`.

Wstrzykiwanie zależności za pomocą konstruktorów

Prześledźmy teraz ten sam przykład, ale zamiast setterów użyjmy argumentu konstruktora do przekazania referencji do obiektu zależnego.

Listing 3.24. Wstrzykiwanie zależności przy pomocy konstruktorów

```
1 <beans>  
2   <bean id="nameProvider" class="przyklady.rozdzial3.NameProvider" />  
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter">  
4     <constructor-arg><ref bean="nameProvider" /></constructor-arg>  
5   </bean>
```

6 | </beans>

Element `<constructor-arg>` służy właśnie do ustawiania argumentów konstruktora. Jeśli konstruktor miałby więcej niż jeden argument to oczywiście należy dla każdego z nich stworzyć odpowiedni wpis `<constructor-arg>`.

Wstrzykiwanie zależności za pomocą konstruktorów demonstruje klasa `przyklady.rozdzial3.ConstructorInjectionExample`.

Efekt działania wstrzykiwania zależności przez konstruktor z pozoru niczym nie różni się od poprzedniego przykładu, w którym użyto do tego celu metod ustawiający pola. I w jednym i w drugim przykładzie został osiągnięty ten sam cel – obiekt `NameWriter` otrzymał referencję do obiektu `NameProvider`. Niemniej jednak warto wspomnieć o pewnej istotnej różnicy wynikającej z zastosowania innego sposobu wstrzykiwania zależności. Otóż używając wstrzykiwania zależności za pomocą konstruktorów możemy mieć pewność, że bean, którego chcemy użyć został poprawnie zainicjowany. Jeśli np. nie byłoby konstruktora domyślnego w klasie a programista zapomniałby dodać odpowiedniego elementu `<constructor-arg>` do pliku konfiguracyjnego to obiekt w ogóle nie zostałby utworzony. Wirtualna maszyna zgłosiłaby stosowny wyjątek i kontener IoC w ogóle by nie wystartował. W przypadku wstrzykiwania zależności za pomocą setterów obiekt zostałby utworzony, kontener IoC funkcjonowałby z pozoru poprawnie, a o błędnej konfiguracji zostalibyśmy poinformowani dopiero podczas próby wywołania metody biznesowej źle skonfigurowanego beanu.

O tym jak radzić sobie ze sprawdzaniem i zapewnieniem poprawnej konfiguracji beanów w dalszej części tego rozdziału.

Co potrafi, a czego nie kontener IoC?

W powyższym prostym przykładzie sytuacja była wręcz komfortowa. Dwa beany – prosta zależność. Taki scenariusz jednak daleko odbiega od typowego zastosowania. Często zależności są dużo bardziej zawile, kaskadowe, jeden bean często wymaga do poprawnego funkcjonowania wiele innych beanów.

Spring Framework umożliwia zdefiniowanie prawie dowolnych zależności między obiektami. Jedynym wyjątkiem są zależności cykliczne, gdy bean **A** zależy od beana **B** i jednocześnie bean **B** zależy od beana **A**. Takiej sytuacji kontener IoC w Spring Framework nie potrafi obsłużyć, ale potrafi ją wykryć i zasygnalizować błąd w konfiguracji.

3.2.2. Sprawdzanie poprawności konfiguracji beanów

Poprawne skonfigurowanie beanów to klucz do prawidłowego działania aplikacji. Nic nie jest bardziej frustrujące dla programisty jak nagle pojawiające się w logach `java.lang.NullPointerException`. . . Jak bronić się przed takimi sytuacjami?

Błędy literowe są mało dokuczliwe, gdyż zostaną wyłapane już na etapie przetwarzania pliku XML przez parser oraz tworzenia definicji beanów przez kontener IoC Spring Framework.

Więcej kłopotu może sprawiać dodanie nowego settera do beana bez dodania odpowiedniego wpisu w pliku konfiguracyjnym. Tego typu błędu nie da się wykryć na etapie uruchamiania kontenera. W efekcie otrzymamy bean, który co prawda został utworzony, ale kontener nie ustawił w nim wszystkich pól i tym samym bean taki nie jest w stanie prawidłowo realizować swoich zadań.

Przed taką sytuacją można obronić się na dwa sposoby. Pierwszy z nich to rezygnacja ze wstrzykiwania zależności za pomocą setterów i używanie do tego celu jedynie konstruktorów. Pomyłki zostaną wychwycone od razu na etapie tworzenia obiektu. Sposób ten jest z jednej strony niezawodny, ale z drugiej bywa dość niewygodny, zwłaszcza jak zależności jest dużo. Trudno nazwać przejrzystym i ładnym konstruktor, który ma np. dziesięć, czy więcej argumentów. . .

Spring Framework dostarcza alternatywne rozwiązanie umożliwiające sprawdzenie poprawności beana po jego utworzeniu. poznaliśmy je już przy okazji omawiania cyklu życia beanów w rozdziale 3.1.2. Przypomnijmy: implementując interfejs `InitializingBean` lub dodając do definicji beana

atrybut `init-method` możemy sprawdzić, czy wszystkie pola zostały poprawnie ustawione i czy bean jest gotowy do świadczenia usług. Jeśli tak nie jest to mamy szansę zgłosić stosowny wyjątek. Bardzo wcześnie (na etapie tworzenia kontenera) jesteśmy więc w stanie wykryć braki w konfiguracji.

3.2.3. Automatyczne dopasowywanie zależności

Tworzenie pliku konfiguracyjnego fabryki beanów jest dość uciążliwe, zwłaszcza w przypadku dużych fabryk zarządzających np. kilkudziesięcioma zależnymi beanami. Liczba beanów pomnożona przez liczbę właściwości, które należałoby zadeklarować może przełożyć się na bardzo duży plik XML, który jest trudny do edycji. Pojawia się więc oczywiste pytanie, czy nie da się uprościć pliku konfiguracyjnego? Czy np. dopasowywanie zależności nie mogłoby się odbywać automatycznie, tak jak ma to miejsce np. w przypadku innego popularnego kontenera IoC – Picocontainera?

Spring Framework jest pod tym względem bardzo elastyczny i oferuje możliwość automatycznego wyszukiwania obiektów zależnych w fabryce. Można tego dokonać na kilka sposobów:

- beany można dopasowywać na podstawie ich nazw,
- beany można dopasowywać na podstawie ich typów,
- beany można dopasowywać na podstawie ich konstruktorów.

Automatyczne dopasowywanie beanów na podstawie nazw

Pierwszy sposób automatycznego dopasowywania beanów opiera się na bardzo prostym założeniu. Jeżeli jeden bean posiada właściwość `xxx` w myśl specyfikacji Java Beans (czyli posiada publiczną metodę `void setXxx()`), to kontener IoC poszuka beana o nazwie `xxx` i przekaże referencję do niego. To najprostszy i chyba najbardziej intuicyjny sposób automatycznego definiowania zależności. W pliku konfiguracyjnym wygląda to następująco:

Listing 3.25. Dopasowywanie zależności wg nazwy

```
1 <beans>
2   <bean id="nameProvider" class="przyklady.rozdzial3.NameProvider" />
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter" autowire="byName" />
4 </beans>
```

Klasa `NameWriter` posiada publiczną metodę `setNameProvider()`, zastosowaliśmy dopasowanie wg nazwy (atrybut `autowire="byName"`), dlatego też kontener poszuka wśród beanów jednego o nazwie `nameProvider`, zainicjuje go i przekaże referencję do niego beanowi `NameWriter`.

Automatyczne dopasowywanie beanów na podstawie typu

Innym typem automatycznego dopasowywania beanów jest dopasowywanie ich na podstawie typu. Listing 3.26 przedstawia przykładową konfigurację:

Listing 3.26. Dopasowywanie zależności wg typu

```
1 <beans>
2   <bean id="dowolnaNazwa" class="przyklady.rozdzial3.NameProvider" />
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter" autowire="byType" />
4 </beans>
```

Atrybut `autowire="byType"` informuje fabrykę beanów, że powinna ona poszukać w klasie `NameWriter` wszystkich publicznych metod ustawiających, sprawdzić typy argumentów, jakie te metody przyjmują, a następnie wyszukać pasujące do tych typów beany zadeklarowane w kontenerze.

Dopasowywanie na podstawie typu ma jedną zasadniczą wadę: może być stosowane tylko w przypadku, gdy mamy pewność, że istnieje tylko jeden bean szukanego typu. Jeśli jest ich więcej kontener zgłosi wyjątek i zakończy pracę.

Automatyczne dopasowywanie beanów na podstawie konstruktorów

Trzecim typem automatycznego dopasowywania beanów jest wyszukiwanie ich na podstawie argumentów konstruktora. Zasada działania jest tu analogiczna do dopasowywania wg typu, z tym jednak wyjątkiem, że pod uwagę brane są argumenty konstruktorów zamiast setterów. Listing 3.27 zawiera przykładową konfigurację:

Listing 3.27. Dopasowywanie zależności wg konstruktorów

```
1 <beans>
2   <bean id="nameProvider" class="przyklady.rozdzial3.NameProvider" />
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter"
4     autowire="constructor" />
5 </beans>
```

Pełen automat

Spring Framework oferuje również jeden specjalny tryb automatycznego wyszukiwania zależności. Jeśli wartość atrybutu `autowire` ustawimy na `autodetect` to kontener spróbuje dobrać optymalną metodą szukania zależności. Algorytm przedstawia się następująco:

- jeśli klasa nie posiada domyślnego konstruktora to zależności dopasowywane są na podstawie dostępnego konstruktora, jeśli klasa posiada kilka konstruktorów to wybierany jest ten z największą ilością atrybutów,
- jeśli natomiast klasa posiada domyślny konstruktor to stosowana jest metoda automatycznego dopasowywania zależności na podstawie typu (`autowire="byType"`).

Domyślna polityka autodopasowania

Z przytoczonych przykładów wynika, że atrybut `autowire` należy dodać do każdego beana, którego autodopasowanie ma dotyczyć. Domyślną polityką Spring Framework jest całkowity brak autodopasowania (atrybut `autowire` przyjmuje wartość `no`). Jeśli chcielibyśmy, aby wszystkie beany w kontenerze podlegały np. regule autodopasowywania wg nazwy powinniśmy przy każdej deklaracji beanu umieścić atrybut `autowire` i nadać mu pożądaną wartość.

Ręczne przerabianie całego pliku konfiguracyjnego mogłoby być dość monotonne i czasochłonne. Autorzy Spring Framework przewidzieli więc możliwość ustawienia domyślnej polityki autodopasowania dla całego kontenera. Wystarczy ustawić atrybut `default-autowire` głównego elementu

beans nadając mu jedną wartość z `no`, `byName`, `byType`, `constructor` lub `autodetect`. Np.:

Listing 3.28. Domyślna polityka autodopasowania

```
1 <beans default-autowire="byName">
2   <bean id="nameProvider" class="przyklady.rozdzial3.NameProvider" />
3   <bean id="nameWriter" class="przyklady.rozdzial3.NameWriter" />
4 </beans>
```

spowoduje, że wszystkie beany będą podlegać regule dopasowywania zależności wg nazwy.

Stosować autodopasowywanie czy nie?

Autorzy Spring Framework nie polecają stosowania mechanizmu automatycznego dopasowywania zależności. Mimo kilku niewątpliwych zalet (znaczące zmniejszenie wielkości pliku XML, czy brak konieczności wprowadzania zmian w pliku XML w przypadku zmiany sygnatury konstruktora, czy dodania nowych setterów) stosowanie autodopasowania może wprowadzić sporo zamieszania. Przede wszystkim nie jest oczywiste dla czytającego tak skonstruowany plik konfiguracyjny. Aby zrozumieć zależności między beanami trzeba zajrzeć do kodu. Oczywiście prawdą jest, że prosty, zrozumiały, intuicyjny i samodokumentujący się kod to klucz do sukcesu projektu w dłuższej perspektywie. Automatyczne dopasowywanie wprowadza do projektu swego rodzaju magię - coś się dzieje, ale nie widać dlaczego.

Oczywiście istnieją sytuacje, gdzie stosowanie autodopasowywania jest wręcz wskazane. Są to przede wszystkim małe projekty (prototypy), które pisze się szybko i wprowadza częste zmiany w kodzie. Autodopasowywanie uwalnia wtedy programistę od konieczności częstej ręcznej modyfikacji pliku XML.

Z doświadczenia mogę powiedzieć, że całkiem nieźle sprawdza się dopasowywanie wg nazwy. Przede wszystkim dlatego, że wymusza pewną konwencję nazewniczą (nazwa beana musi być tożsama z setterem), co zapewnia, że kod staje się bardzo intuicyjny - wiemy, czego się spodziewać i gdzie. Jeśli dodatkowo zastosuje się samoopisujące się nazwy setterów (np.

`nameProvider` zamiast `nProv`) to czytelny pozostanie zarówno kod, jak i plik konfiguracyjny.

Problematyczne natomiast może okazać się dowiązywanie wg typu, ponieważ na jakimś etapie prac nad projektem może się zdarzyć, że pojawią się dwa lub więcej beany tego samego typu. To z kolei doprowadziłoby pewnie do sytuacji, w której autodopasowanie stosowane by było dla większości beanów (bo tak np. ustawiona byłaby domyślna polityka `autowire`), a dla kilku należałoby zdefiniować zależności *explicite*. Czytelność takiego pliku drastycznie się pogarsza.

Jak w każdym przypadku złoty środek należy znaleźć samemu. Początkującym użytkownikom Spring Framework zalecam nie korzystanie z mechanizmu autodopasowania.

ROZDZIAŁ 4

Programowanie aspektowe w Spring Framework

Co jakiś czas pojawiają się w informatyce przełomowe koncepcje, które rzucają zupełnie nowe światło na dotychczas stosowane rozwiązania. Jednym z większych skoków jakościowych ostatnich lat jest, zdaniem autora i nie tylko, programowanie aspektowe (*ang. AOP - Aspect Oriented Programming*). Pozwala ono spojrzeć na program w kategoriach powtarzających się zadań (czyli właśnie aspektów), które można wydzielić do postaci niezależnych od siebie modułów i połączyć wzajemnie tam, gdzie ich funkcje się krzyżują. Kod źródłowy staje się przez to dużo prostszy, bardziej elegancki, a jednocześnie zachowuje w pełni swoją funkcjonalność.

W niniejszym rozdziale omówione zostaną koncepcja programowania aspektowego oraz mechanizmy realizujące AOP w Spring Framework.

Osoby, które teoretyczne podstawy AOP mają już opanowane, mogą pominąć lekturę następnego podrozdziału i przejść bezpośrednio do podrozdziału 4.3.

4.1. Wstęp do aspektów

Aby nadać naszym rozważaniom konkretny kształt przyjrzyjmy się prostemu przykładowi, swoistemu „Hello World” w świecie aspektów. Załóżmy, że pewna aplikacja zawiera usługę (metodę), której wywołania chcielibyśmy zalogować w dzienniku zdarzeń. Interesowałby nas konkretnie czas wykonywania się pewnej metody.

Stwórzmy usługę wraz z interesującą nas metodą:

Listing 4.1. Usługa, której wywołania chcemy logować

```
1 package przyklady.rozdzial4;
2
3 public interface IService {
4     void service();
5 }
6
7 package przyklady.rozdzial4;
8
9 public class Service implements IService {
10     public void service() {
11         // metoda biznesowa
12     }
13 }
```

Metoda `service()` jest używana w różnych miejscach projektu. Aby wykonać nasze zadanie możemy dodać do kodu aplikacji odpowiednie linijki:

Listing 4.2. Rozwiązanie oczywiste

```
1 System.out.println("Wchodze do metody service(): " + System.currentTimeMillis());
2 service();
3 System.out.println("Wychodze z metody service(): " + System.currentTimeMillis());
```

To działa, ma jednak jedną zasadniczą wadę. Może się zdarzyć, że w bardzo krótkim czasie staniemy się posiadaczami kodu, w którym ilość linijek diagnostycznych będzie większa niż kodu właściwego. Powinniśmy więc poszukać lepszego rozwiązania, które nie przyczyniałoby się do zaciemnienia kodu głównego.

Najprostszym rozwiązaniem, jakie każdemu zapewne przychodzi na myśl jest umieszczenie linijki wypisującej komunikat bezpośrednio w ciele metody `service()`. Jest to już zdecydowanie lepsze rozwiązanie. Kod wypisujący komunikaty znajduje się w jednym miejscu i wszystko wydaje się być w porządku... do czasu, gdy rozbudujemy naszą usługę o kolejną metodę biz-

nesową np. `service2()`, której wywołania również chcielibyśmy ujrzyć w dzienniku systemowym.

Dodanie linijek `System.out...` do każdej nowej metody biznesowej nie rozwiązuje globalnie problemu, przenosi go jedynie w inne miejsce.

Trzeba poszukać więc jeszcze lepszego, definitywnego rozwiązania.

4.1.1. Obiekty Proxy

JDK od wersji 1.3 oferuje możliwość tworzenia specjalnych obiektów, tzw. dynamicznych pośredników (ang. *dynamic proxy*). Nawet pobieżna lektura API klasy `java.lang.reflect.Proxy` powinna wystarczyć, aby uznać ją za godną kandydatkę do rozwiązania naszego problemu. Dlaczego? Ponieważ umożliwia wykonywanie metod nie bezpośrednio na obiektach, ale na pośrednikach, które z kolei delegują (o ile tego chcą) wywołanie do obiektu docelowego. Gdyby więc ustanowić obiekt pośredniczący dla naszej usługi to może dałoby się w obiekcie pośredniczącym zaszyć jakieś sprytne logowanie wywoływania metod? Sprawdźmy to!

Listing 4.3. Dynamiczny pośrednik

```
1 package przyklady.rozdzial4;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
6
7 public class DynamicProxySample {
8
9     public static void main(String[] args) throws Exception {
10         new DynamicProxySample().run();
11     }
12
13     public void run() {
14         IService svc = (IService) Proxy.newProxyInstance(
15             IService.class.getClassLoader(),
16             new Class[] { IService.class },
17             new LoggerHandler(new Service()));
18         svc.service();
19         svc.service2();
20     }
21
22     private class LoggerHandler implements InvocationHandler {
23
24         private Object target;
25
26         public LoggerHandler(Object target) {
27             this.target = target;
28         }
29     }
30 }
```

```
30
31     public Object invoke(Object proxy, Method method,
32         Object[] args) throws Throwable {
33         System.out.println("+ " + method.getName() + " " + time());
34         Object returnValue = method.invoke(target, args);
35         System.out.println("- " + method.getName() + " " + time());
36         return returnValue;
37     }
38
39     private long time() {
40         return System.currentTimeMillis();
41     }
42 }
43 }
```

Listing 4.3 pokazuje sposób, w jaki obiekty proxy umożliwiają grupowanie pewnych problemów i kompleksowe ich rozwiązywanie.

W wierszach 14 – 17 tworzony jest obiekt pośrednika, który implementuje interfejs `IService`. Ostatni argument metody `newProxyInstance()` pozwala na przekazanie nowotworzonemu pośrednikowi kodu, który powinien zostać wykonany w momencie wywołania dowolnej metody ma obiekcie pośrednika. Ścile rzecz ujmując jest to dowolna klasa implementująca interfejs `InvocationHandler`. W naszym przykładzie jest to klasa zdefiniowana w wierszach 23 – 42.

Jako argument konstruktora klasa `LoggerHandler` przyjmuje dowolny obiekt. Zatem przekazując w konstruktorze obiekt klasy `Service` możemy w metodzie `invoke(...)` kontrolować wywołania metod naszej usługi. Nasze nowe proxy będzie delegowało wszystkie wywołania metod do usługi właściwej, a przy okazji możemy dopisać linijki diagnostyczne, o które nam chodziło. Dzięki temu prostemu zabiegowi „udekorowaliśmy” naszą usługę wzbogacając nieco jej funkcjonalność.

W ten sposób udało się całkiem sporo osiągnąć. Kod programu nie zawiera dodatkowych linijek, kod usługi pozostał nietknięty, loguje wywołania metod niezależnie od ich ilości - po dodaniu nowych zachowa się dokładnie tak, jak tego oczekujemy. Jedyne, co musieliśmy zrobić to skorzystać z obiektu pośrednika i napisać fragment kodu realizujący pożądaną przez nas funkcjonalność.

Idea dynamicznych pośredników to podstawa, na której zbudowany został framework AOP w Spring Framework. Zanim jednak przejdziemy do omówienia

szczegółów tej części Spring Framework warto poświęcić jeszcze kilka chwil samemu aspektom.

4.2. Aspekty pod lupą

4.2.1. Koncepcja AOP

Przeanalizujmy, co skłoniło nas do użycia obiektu proxy?

Przede wszystkim zidentyfikowaliśmy pewne powtarzające się zadanie. Zauważyliśmy, że zadanie to wymaga ciągłego stosowania podobnego lub wręcz identycznego kodu. Na dodatek kod ten jest funkcjonalnie niezależny od logiki aplikacji właściwej (nasza przykładowa aplikacja będzie bez niego działać poprawnie, choć nie będzie wypisywała komunikatów).

Aspekty

W terminologii AOP powiedzielibyśmy, że wyodrębniliśmy pewien **aspekt**, który być może wymaga szczególnego potraktowania.

Aspekt jest więc wydzieloną funkcjonalnie częścią programu, pewnym modulem, który możemy bezkolizyjnie i bezinwazyjnie połączyć z innymi modułami (aspektami). Realizuje on określone zadanie i koncentruje się tylko na problemie (ang. *concern*), którego ściśle dotyczy (np. trzymając się naszego przykładu będzie to np. logowanie wywoływania metod).

Wszędzie tam, gdzie mamy do czynienia z zazębaniem, czy przecinaniem się pewnych zadań możemy zastosować aspekty w miejsce tradycyjnego tworzenia hierarii klas i zawikłych powiązań między nimi.

Problem przecinania się zadań (ang. *cross-cutting concerns*) dotyczy niemal każdej aplikacji. Np. aplikacja musi być zarówno wydajna, jak i bezpieczna (wydajność i bezpieczeństwo można potraktować jako przykłady aspektów). Zadania te można sobie wyobrazić jako pewne płaszczyzny, które dopiero w aplikacji znajdują pewien punkt przecięcia - spotykają się, by wspólnie realizować kompleksowe zadanie. Aspekty i AOP sprawiają, że

realizacja takich zadań jest prosta, znacznie prostsza niż w tradycyjnym obiektowym podejściu, opartym na hierarchii klas.

Aspekty umożliwiają lepszą enkapsulację (usługa biznesowa nie musi wiedzieć, czy logowane są wywołania jej metod, a moduł logujący nie musi wiedzieć, co właściwie loguje) oraz zwiększają w znacznym stopniu ponowne wykorzystanie raz napisanego kodu (nasz aspekt, jako samodzielnie funkcjonujący moduł, możemy zastosować z dowolną inną usługą biznesową).

Instrukcje

W jaki sposób dodaliśmy nowy aspekt do kodu naszej usługi? Użyliśmy obiektu proxy, który otoczył wywołanie każdej metody biznesowej usługi pewnym dodatkowym kodem. Ów kod nazwany jest w świecie aspektów **instrukcją** (ang. *advice*).

W dalszej części tego rozdziału przekonamy się, że istnieje kilka typów instrukcji, których możemy używać w zależności od potrzeb. Dzięki instrukcjom możemy wpływać na przebieg wykonywania programu poprzez wzbogacenie kodu, podmienianie go, albo odpowiednie reagowanie, w przypadku pojawienia się wyjątku.

Punkty złączeń

Punkt złączenia (ang. *join point*) to dowolne miejsce w kodzie programu głównego (w naszym przykładzie jest to kod usługi), w którym styka się on z aspektem. Punktem złączenia w naszej aplikacji jest wywołanie metody biznesowej. W tym miejscu aplikacja powinna oprócz własnego kodu wykonać również kod instrukcji (jednej lub więcej).

Wywołanie metody to pewnie najprostszy, ale oczywiście nie jedyny możliwy punkt złączenia. Inne przykłady to m.in.:

- wywołanie konstruktora pewnej klasy,
- dostęp do pola,

- wykonanie jakiegoś szczególnego kawałka kodu,
- wystąpienie wyjątku,
- etc.

Punkt przecięcia

Ostatnim ważnym pojęciem z zakresu AOP jest **punkt przecięcia** (ang. *pointcut*), który jest niczym innym, jak tylko zbiorem punktów złączeń. Punkt przecięcia określa, w których miejscach aspekt spotyka się z kodem głównym. W przypadku naszej przykładowej usługi punktem przecięcia są wszystkie jej metody.

Różne narzędzia AOP stosują różne notacje umożliwiające definiowanie punktów przecięć. Jednym z powszechniej stosowanych sposobów jest użycie wyrażeń regularnych celem wskazania metod należących do danego punktu przecięcia. Np. wyrażenie `set*` oznaczać może wszystkie metody ustawiające (settery).

4.2.2. Sposoby implementacji AOP

W jaki sposób można zaimplementować AOP?

Istnieją przynajmniej dwie powszechnie stosowane metody implementacji paradygmatu AOP. Przyjrzyjmy się dokładniej, na czym one polegają.

Obiekt proxy

Podstawowym i najbardziej naturalnym narzędziem, które już poznaliśmy, jest **obiekt proxy**. Niezaprzeczalną zaletą tego rozwiązania jest jego prostota i wsparcie ze strony samego języka Java. Podejście to nie jest jednak pozbawione wad, z których najważniejszą jest brak możliwości tworzenia obiektów proxy dla konkretnych klas. Język Java przewiduje tylko i wyłącznie tworzenie obiektów proxy dla interfejsów.

Drugą ciemną stroną stosowania obiektów dynamicznych pośredników jest ich niższa niż „czystych” klas wydajność. Związany z utworzeniem i działaniem obiektu proxy dodatkowy narzut czasowy, może, przy bardzo częstych wywołaniach metod, nie być obojętny dla ogólnej wydajności aplikacji¹.

Generowanie byte-code’u

Nie zawsze stosowanie prostych obiektów proxy jest wystarczające. Nie trudno wyobrazić sobie sytuację, gdy usługa nie implementuje żadnego interfejsu i nie mamy jej kodu źródłowego (np. jest to zakupiona komercyjna, zamknięta biblioteka). Czy można w takiej sytuacji zastosować AOP?

Można, choć trzeba uciec się do nieco bardziej wyrafinowanej metody, a mianowicie sztuczki określanej mianem generowania byte-code’u. Korzystając z pomocy takich bibliotek jak np. CGLIB (<http://cglib.sf.net>) można „w locie” wygenerować kod binarny klasy, która oprócz własnej funkcjonalności zostanie wzbogacona o funkcjonalność pochodzącą z kodu instrukcji.

Ten sposób nie tylko eliminuje problem wydajności, ale również pozwala w dużym stopniu rozszerzyć możliwości aspektów. Wreszcie możliwe staje się:

- stworzenie pośrednika dla konkretnej klasy, a nie tylko dla interfejsu,
- dodanie interfejsów do klas, które pierwotnie ich nie implementują,
- definiowanie punktów złączeń np. wewnątrz metod.

Wariacje na temat AOP

Wyżej wymienione narzędzia realizacji AOP występują w różnych wariantach. Niektóre projekty realizują AOP na poziomie instancji, inne na poziomie całej klasy. Ten drugi przypadek wymaga zastosowania dedykowanej

¹choć trzeba przyznać, że z każdym nowym wydaniem JVM wydajność tego rozwiązania rośnie

ładowarki klas (ang. *classloader*), która w miejsce zwykłej instancji klasy utworzy instancję wzbogaconą o kod instrukcji.

Innym przykładem może być wygenerowanie wzbogaconego kodu Java, jeszcze przed kompilacją programu, choć od tego rozwiązania konsekwentnie się odchodzi ze względu nie niewygodę stosowania i możliwość osiągnięcia tych samych efektów przy znacznie mniejszych nakładach pracy.

4.2.3. Zastosowania AOP

Opanowawszy teoretyczne podstawy programowania aspektowego (szczegóły poznamy za chwilę) zastanówmy się do czego można wykorzystać tę koncepcję w praktyce? Jakie profity może przynieść stosowanie AOP typowej aplikacji? Do czego AOP jest przeważnie stosowany?

Diagnozowanie i monitorowanie

Mieliśmy już okazję poznać jedno zastosowanie aspektów, czyli dodawanie do kodu linijek diagnostycznych informujących o stanie aplikacji. W ten sposób można monitorować wydajność i stan programu.

Przewaga aspektów nad specjalistycznymi bibliotekami do logowania zdarzeń polega na tym, że kod bazowy aplikacji nie zawiera dodatkowych, zaciemniających algorytm linii. Logowanie zdarzeń realizowane jest w sposób przezroczysty. Co więcej, aspekt diagnostyczny można np. wyłączyć w produkcyjnej instalacji aplikacji (jeśli powodowałby zbyt duże obciążenie), czy też zamienić go na, bardziej w takiej sytuacji przydatny, aspekt monitorujący.

Kontrola dostępu

Wyobraźmy sobie sytuację, w której dostęp do wybranych części aplikacji wymaga specjalnych uprawnień, np. użytkownik powinien być zalogowany i posiadać odpowiednią rolę. Jak to osiągnąć?

W tradycyjnym podejściu należałoby przed wejściem lub tuż po wejściu do chronionych metod sprawdzić czy spełnione są warunki bezpieczeństwa.

Każda chroniona metoda posiadałaby podobny lub identyczny kod gwarantujący spełnienie wymogów bezpieczeństwa.

Dzięki aspektom możliwe jest wydzielenie takiego kodu „wartownika” do swoistej czarnej skrzynki, a poprzez odpowiednie zdefiniowanie punktów przecięć proste staje się rozpięcie parasola ochronnego nad wszystkimi chronionymi fragmentami kodu bazowego. Co ważne, dzieje się to w sposób całkowicie transparentny dla aplikacji.

Transakcje

Kolejne, bodaj najpowszechniejsze, zastosowanie AOP to wydzielenie do niezależnego modułu całego kodu związanego z zarządzaniem transakcjami.

W przypadku transakcji bazodanowych oznacza to, że kod bazowy aplikacji wykonuje tylko i wyłącznie zapytania SQL związane z operacjami na rekordach (bezpośrednio lub z wykorzystaniem bibliotek O/R), nie przejmując się szczególnie współbieżnością. Transakcyjność zapewniana jest przez odpowiednio skonstruowany aspekt.

Punktami złączeń będą w takim przypadku wszystkie metody `load(...)`, `save(...)`, `delete(...)`, *etc.*, które dokonują trwałych modyfikacji danych. Metody te staną się dzięki aspektowi dużo czytelniejsze, nie będą zawierały niekończących się bloków try `{BEGIN ... COMMIT} catch {ROLLBACK}`. Będą tylko starały się wykonać swoje podstawowe zadanie, a zapewnienie wyłącznego dostępu do źródła danych i stosowną reakcję na ewentualne wyjątki zapewni aspekt.

Pamięć podręczna

Aspektów można również użyć do dodania do aplikacji pamięci podręcznej (ang. *cache*), przyspieszającej jej działanie.

Aplikując odpowiedni aspekt w kodzie bazowym można przechwycić wywołanie niektórych czasochłonnych metod, sprawdzić argumenty wejściowe metody, a następnie poszukać w pamięci podręcznej, czy dla tych argumentów nie została już wcześniej wyliczona i zapamiętana wartość wynikowa.

Jeśli tak, to można zwrócić tą wartość bezpośrednio, bez wykonywania czasochłonnego algorytmu.

Znowu może dziać się to w sposób niezauważalny dla aplikacji.

Inne zastosowania

Zastosowań aspektów może być oczywiście znacznie więcej. Przytoczone przykłady służą jedynie do lepszego zobrazowania całej idei i nadania jej realnego kształtu.

Dzięki aspektom możliwe staje się pisanie przejrzystych, eleganckich i modularnych aplikacji, dlatego warto rozważyć to podejście przed napisaniem większego fragmentu kodu - być może AOP stanowić będzie najprostsze i optymalne rozwiązanie.

Nawet jeśli nie wiemy dokładnie, w jakim kierunku tworzona przez nas aplikacja będzie w przyszłości ewoluować, aspekty mogą okazać się jedyną drogą do szybkiego dodawania nowej funkcjonalności, bez potrzeby wprowadzania rewolucyjnych zmian w kodzie bazowym.

4.2.4. Frameworki AOP

Framework AOP to zbiór narzędzi, których zadaniem jest umożliwienie stosowania paradygmatu programowania aspektowego w sposób możliwie prosty, efektywny, powtarzalny i niewidoczny dla bazowego kodu aplikacji.

W praktyce frameworki AOP to specjalne biblioteki, które obudowują podstawowe narzędzia realizacji aspektów (proxy oraz generowanie byte-code'u) w dodatkowe, ułatwiające ich używanie funkcje, takie jak np. elastyczna konfiguracja (zaawansowane możliwości definiowania punktów przecięć np. w czytelnych plikach XML), czy repozytoria gotowych do użycia, często powtarzających się w aplikacjach instrukcji.

AspectJ

Żadna książka, w której poruszana jest tematyka programowania aspektowego, nie może pominąć znaczenia pierwszego i najważniejszego projektu AOP, jakim zapewne jest AspectJ².

AspectJ jest najpopularniejszym frameworkiem AOP, stworzonym m.in. przez samego autora koncepcji AOP Gregora Kiczalesa. Jest to najprawdopodobniej najbardziej zaawansowany projekt tego typu, oferujący najszersze spektrum możliwości.

AspectJ rozszerza język Java o dodatkowe słowa kluczowe i składnię umożliwiającą definiowanie aspektów. Kod napisany w AspectJ jest integrowany z byte-codem aplikacji bazowej w procesie określanym mianem *weaving*. *Weaving* polega na zlokalizowaniu na podstawie definicji zawartych w kodzie AspectJ punktów przecięć i odpowiednim zmodyfikowaniu byte-code'u aplikacji bazowej. W wyniku tego procesu generowany jest nowy byte-code, wzbogacony o kod pochodzący z instrukcji.

Jeśli *weaving* zostanie wykonany tuż po skompilowaniu klas aplikacji bazowej i przed uruchomieniem JVM to mówimy o kompilacji statycznej. Drugim sposobem jest manipulowanie byte-codem już w czasie pracy aplikacji, co określane jest mianem *load-time weaving*.

Stworzenie rozszerzenia języka Java i dodatkowy krok kompilacji, jakim jest *weaving*, mogą być postrzegane jako największa wada AspectJ. Autorzy projektu zadbali jednak o to, aby tworzenie aspektów nie było dla programisty zbyt uciążliwe. Dostępna jest wtyczka do środowiska Eclipse – AspectJ Development Tools³ (AJDT), dzięki której korzystanie z AspectJ staje się prostsze.

Osoby szerzej zainteresowane zagadnieniem AOP powinny koniecznie zapoznać się z tym projektem.

²<http://eclipse.org/aspectj>

³<http://www.eclipse.org/ajdt/>

Inne projekty AOP

Podobnie jak wiele innych dziedzin AOP również przeszedł przez fazę bujnego rozkwitu. Nie działa się to może na aż tak wielką skalę, jaką mieliśmy (i mamy nadal!) okazję podziwiać w przypadku frameworków do tworzenia aplikacji internetowych, ale i tak świat Javy doczekał się co najmniej kilkunastu projektów dedykowanych AOP.

Spośród bardziej znanych warto wymienić:

- AspectWerkz - <http://aspectwerkz.codehaus.org/>,
- Nanning - <http://nanning.codehaus.org/>,
- JBossAOP - <http://www.jboss.org/products/aop>,
- Spring Framework AOP

Wszystkie wyżej wymienione rozwiązania stosują odmienne podejście niż AspectJ. Nie rozszerzają one języka Java, ani też nie tworzą własnego. Operują wyłącznie na czystych obiektach POJO, kod wskazówek to zwykle klasy, a konfiguracja AOP przeważnie realizowana jest w pliku XML. Są to więc rozwiązania łatwiejsze w użyciu, gdyż nie wymagają żadnych dodatkowych narzędzi.

Ostatnie dwa projekty (JBossAOP oraz Spring Framework AOP) wydają się zdobywać coraz większą popularność, co po części wynika z tego, że związane są ściśle z bardzo popularnymi na rynku produktami. Oczywiście nie jest to główny powód ich szerokiego stosowania - w rzeczywistości są to potężne narzędzia, które, jak się przekonamy podczas omawiania Spring Framework AOP, umożliwiają osiągnięcie zdumiewających rezultatów przy jednoczesnym zachowaniu przejrzystości kodu.

Po tym, nieco długim, wstępnie możemy przystąpić do zagłębienia się w AOP w Spring Framework, który, zaraz po beanach i kontenerze IoC, jest trzecim najważniejszym składnikiem tej platformy.

4.3. AOP w Spring Framework

Framework AOP w Spring Framework to rozwiązanie czysto Javowe. Nie wymagana jest więc specyficzna składnia do tworzenia aspektów, jak to ma miejsce w AspectJ, nie ma też oddzielnego procesu *weavingu*. Oczywiście taka decyzja twórców Spring Framework pociąga za sobą pewne konsekwencje, z których najistotniejszą jest ta, że framework ten nie umożliwia implementacji wszystkich koncepcji, które mogą być z powodzeniem zrealizowane przy użyciu AspectJ. Była to jednak decyzja świadoma. Sami twórcy Spring Framework określają swój AOP jako:

- narzędzie, które w połączeniu z kontenerem IoC ma w prosty sposób rozwiązywać **typowe** problemy aplikacji klasy enterprise,
- narzędzie umożliwiające korzystanie w programie z różnych usług enterprise (np. EJB, JTA) w sposób deklaracyjny.

To pragmatyczne podejście dało w efekcie bardzo proste i elastyczne w użyciu narzędzie, które w połączeniu z gotowymi klasami rozwiązującymi typowe problemy aplikacji staje się wręcz niezastąpione.

Użytkownikowi, który potrzebuje tylko podstawowych funkcji AOP, wystarczy kilkanaście minut, żeby móc samodzielnie definiować aspekty w Spring Framework. Zobaczmy jak wyglądałaby nasza przykładowa usługa, zrealizowana z użyciem Spring IoC i Spring AOP.

4.3.1. `org.springframework.aop.framework.ProxyFactoryBean`

`ProxyFactoryBean` to specjalny bean, który potrafi utworzyć instancję dynamicznego pośrednika dla dowolnego obiektu. Jest to więc podstawowe narzędzie realizacji AOP w Spring Framework.

Podsumowanie

Czas na podsumowanie

Listings

3.1. Pierwszy bean – HelloWorld	17
3.2. Minimalny plik konfiguracyjny w wersji XML	17
3.3. Minimalny plik konfiguracyjny w wersji *.properties	17
3.4. Przykład wykorzystania beanu HelloWorld	18
3.5. Bardziej szczegółowy plik konfiguracyjny	18
3.6. Klasa bez konstruktora domyślnego	20
3.7. Konfiguracja właściwości beanów	21
3.8. Definiowanie kolekcji	21
3.9. Metody zgodne ze specyfikacją JavaBean	22
3.10. Zagnieżdżony bean	22
3.11. Interfejs InitializingBean	24
3.12. Interfejs DisposableBean	24
3.13. Deklaratywne definiowanie cyklu życia	25
3.14. Przykład konfiguracji beanu z polem typu java.util.Date	26
3.15. Metoda ustawiająca pole date	26
3.16. Rejestrowanie własnych edytorów właściwości	26
3.17. Klasa bez konstruktora domyślnego	27
3.18. Konfiguracja konstruktora	27
3.19. Interfejs org.springframework.beans.factory.FactoryBean	28

3.20. Fabryki beanów	28
3.21. Przykładowe beany - NameProvider	31
3.22. Przykładowe beany - NameWriter	31
3.23. Wstrzykiwanie zależności przy pomocy setterów	32
3.24. Wstrzykiwanie zależności przy pomocy konstruktorów	32
3.25. Dopasowywanie zależności wg nazwy	35
3.26. Dopasowywanie zależności wg typu	36
3.27. Dopasowywanie zależności wg konstruktorów	37
3.28. Domyślna polityka autodopasowania	38
4.1. Usługa, której wywołania chcemy logować	42
4.2. Rozwiązanie oczywiste	42
4.3. Dynamiczny pośrednik	43

Spis rysunków

Spis tablic