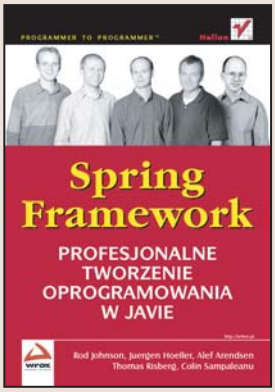
Spring Framework. Profesjonalne tworzenie oprogramowania w Javie



Autorzy: Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu
Tłumaczenie: Piotr Rajca, Mikołaj Szczepaniak
ISBN: 83-246-0208-9
Tytuł oryginału: Professional Java Development with the Spring Framework
Format: B5, stron: 824

Spring to szkielet wytwarzania aplikacji (framework), dzięki któremu proces budowania oprogramowania w języku Java dla platformy J2EE staje się znacznie prostszy i efektywniejszy. Spring oferuje usługi, które można z powodzeniem używać w wielu środowiskach – od apletów po autonomiczne aplikacje klienckie, od aplikacji internetowych pracujących w prostych kontenerach serwletów po złożone systemy korporacyjne pracujące pod kontrolą rozbudowanych serwerów aplikacji J2EE. Spring pozwala na korzystanie z możliwości programowania aspektowego, znacznie sprawniejszą obsługę relacyjnych baz danych, błyskawiczne budowanie graficznych interfejsów użytkownika oraz integrację z innymi szkieletami takimi, jak Struts czy JSF.

Książka „Spring Framework. Profesjonalne tworzenie oprogramowania w Javie" odkryje przed Tobą wszystkie tajniki stosowania Springa w procesie wytwarzania systemów informatycznych dla platformy J2EE. Dowiesz się, jak efektywnie korzystać z najważniejszych składników szkieletu Spring i poznasz ich rolę w budowaniu aplikacji. Nauczysz się nie tylko tego, co można zrealizować za pomocą poszczególnych usług, ale także tego, w jaki sposób zrobić to najlepiej. W kolejnych ćwiczeniach przeanalizujesz proces tworzenie kompletnej aplikacji w oparciu o Spring.

W książce poruszono m.in. tematy:

- Struktura szkieletu Spring
- Tworzenie komponentów i definiowanie zależności pomiędzy nimi
- Testowanie aplikacji i testy jednostkowe
- Programowanie aspektowe w Spring
- Połączenia z relacyjnymi bazami danych za pomocą JDBC
- Zarządzanie transakcjami
- Korzystanie z mechanizmu Hibernate
- Zabezpieczanie aplikacji
- Stosowanie szkieletu Web MVC

Przekonaj się, jak Spring może zmienić Twoją pracę nad tworzeniem aplikacji J2EE

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

2

Fabryka komponentów i kontekst aplikacji

Sercem Springa jest funkcjonalność lekkiego kontenera IoC (ang. *Inversion of Control*). Do skonfigurowania i właściwego powiązania obiektów aplikacji z obiektami frameworka (oraz zarządzania ich cyklami życia) można użyć jednego lub wielu egzemplarzy tego kontenera. Podstawowe cechy kontenera IoC dają nam pewność, że zdecydowana większość tych obiektów nie będzie zawierała zależności wiążących je z samym kontenerem, zatem relacje pomiędzy obiektami można wyrażać za pomocą tak naturalnych rozwiązań (języka Java) jak **interfejsy** czy **abstrakcyjne klasy bazowe**, co niemal w stu procentach uniezależnia nas od sposobu implementacji tych obiektów oraz lokalizacji ich zależności. Okazuje się, że kontener IoC jest podstawą dla bardzo wielu funkcji i mechanizmów, które będziemy analizowali w tym i kolejnych rozdziałach.

Z tego rozdziału dowiesz się, jak konfigurować i korzystać z fabryk komponentów Springa oraz kontekstów aplikacji, czyli dwóch podstawowych składników decydujących o kształcie i funkcjonalności kontenera IoC tego frameworka. Poznasz interfejsy `BeanFactory` i `ApplicationContext` wraz ze wszystkimi ważnymi interfejsami i klasami pokrewnymi, które są wykorzystywane zawsze wtedy, gdy należy programowo utworzyć lub uzyskać dostęp do kontenera IoC. W niniejszym rozdziale skupimy się na tych odmianach interfejsów `BeanFactory` i `ApplicationContext`, które można deklaratywnie konfigurować za pośrednictwem odpowiednich dokumentów XML. Wspomniane interfejsy stanowią podstawową funkcjonalność w zakresie konfigurowania i korzystania z kontenera IoC i są stosowane przez użytkowników Springa w zdecydowanej większości przypadków. Warto jednak pamiętać, że Spring oddziela konfigurację kontenera od mechanizmów jego użytkowania. Podczas lektury kolejnego rozdziału przekonasz się, że dostęp do wszystkich możliwości kontenera można uzyskiwać zarówno za pośrednictwem konfiguracji programowej, jak i alternatywnych formatów deklaratywnych.

W rozdziale zajmiemy się następującymi zagadnieniami:

- odwracaniem kontroli (IoC) i **wstrzykiwaniem zależności**,
- podstawowymi technikami konfigurowania obiektów i definiowania relacji łączących obiekty w ramach kontenera Springa,
- sposobem interpretowania zależności i różnicami pomiędzy jawnym a automatycznym wprowadzaniem zależności,
- obsługą cyklu życia obiektów w kontenerze Springa,
- abstrakcją technik dostępu do usług i zasobów,
- postprocesorami fabryki komponentów i samych komponentów, które odpowiadają za dostosowywanie zachowania kontenera i komponentów,
- technikami programowego korzystania z interfejsów `BeanFactory` i `ApplicationContext`.

Odwracanie kontroli i wstrzykiwanie zależności

W rozdziale 1. opisano koncepcję **odwracania kontroli** (IoC) i wyjaśniono, dlaczego jest ona tak ważna dla procesu wytwarzania oprogramowania. W pierwszej kolejności krótko przypomnimy, co to pojęcie rzeczywiście oznacza, by zaraz potem przystąpić do analizy kilku przykładów jego praktycznego stosowania dla kontenera Springa.

Kod oprogramowania rozbija się zwykle na logiczne, współpracujące ze sobą komponenty lub usługi. W Javie takie komponenty mają przeważnie postać egzemplarzy klas, czyli obiektów. Każdy obiekt realizuje swoje zadania z wykorzystaniem lub we współpracy z pozostałymi obiektami. Przypuśćmy, że mamy dany obiekt *A*; można powiedzieć, że pozostałe obiekty, z których korzysta obiekt *A*, są jego **zależnościami** (ang. *dependencies*). Odwracanie kontroli (IoC) jest w wielu sytuacjach pożądanym wzorcem architekturalnym, który przewiduje, że obiekty są ze sobą wiązane przez byt zewnętrzny (w tym przypadku kontener), który — tym samym — odpowiada za obsługę ich zależności (eliminując konieczność bezpośredniego tworzenia egzemplarzy jednych obiektów w drugich).

Przyjrzyjmy się teraz kilku przykładom.

> *Większość przykładów prezentowanych w tym rozdziale (nawet tych, które nie są prezentowane w formie kompletnych klas czy interfejsów) jest dostępna także w postaci gotowej do kompilacji, z którą możesz swobodnie eksperymentować (patrz ftp://ftp.helion.pl/przyklady/sprifr.zip).*

Przypuśćmy, że dysponujemy usługą obsługującą historyczne dane pogodowe, którą zakodowano w tradycyjny sposób (bez korzystania z kontenera IoC):

```
public class WeatherService {

  WeatherDao weatherDao = new StaticDataWeatherDaoImpl();
```

```
  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

public interface WeatherDao {

  WeatherData find(Date date);
  WeatherData save(Date date);
  WeatherData update(Date date);
}

public class StaticDataWeatherDaoImpl implements WeatherDao {

  public WeatherData find(Date date) {
    WeatherData wd = new WeatherData();
    wd.setDate((Date) date.clone());
    ...
    return wd;
  }

  public WeatherData save(Date date) {
    ...
  }

  public WeatherData update(Date date) {
    ...
  }
}
```

Zgodnie z dobrymi praktykami napisano też przypadek testowy, który weryfikuje prawidłowe funkcjonowanie tego kodu. Jeśli użyjemy popularnego narzędzia JUnit, kod takiego testu może mieć następującą postać:

```
public class WeatherServiceTest extends TestCase {
  public void testSample1() throws Exception {
    WeatherService ws = new WeatherService();
    Double high = ws.getHistoricalHigh(new GregorianCalendar(2004, 0, 1).getTime());
    // ... w tym miejscu powinieneś umieścić dodatkowy kod sprawdzający zwracaną wartość...
  }
}
```

Nasza usługa pogodowa wykorzystuje dane pogodowe w formie obiektu *DAO* (obiektu dostępu do danych; ang. *Data Access Object*), który jest niezbędny do uzyskania danych historycznych. Do obsługi obiektu DAO program wykorzystuje co prawda interfejs `WeatherDao`, jednak w przedstawionym przykładzie usługa pogodowa bezpośrednio tworzy egzemplarz konkretnego, znanego typu DAO, który implementuje ten interfejs: `StaticDataWeather-➥DaoImpl`. Dodatkowo nasza aplikacja testowa, `WeatherServiceTest`, bezpośrednio wykorzystuje konkretną klasę `WeatherService` (eliminuje możliwość specjalizacji). Prezentowany przykład zakodowano bez korzystania z technik IoC. O ile usługa pogodowa współpracuje z odpowiednim obiektem DAO za pośrednictwem interfejsu, konkretny egzemplarz obiektu

DAO (określonego typu) jest przez tę usługę tworzony bezpośrednio i właśnie usługa pogodowa steruje jego cyklem życia; oznacza to, że wspomniana usługa jest obarczona zależnościami zarówno od interfejsu DAO, jak i konkretnej klasy implementującej ten interfejs. Co więcej, przykładowa aplikacja testowa reprezentuje klienta tej usługi, który bezpośrednio tworzy egzemplarz konkretnego typu usługi pogodowej, zamiast korzystać z pośrednictwa właściwego interfejsu. W rzeczywistej aplikacji bardzo prawdopodobne byłoby występowanie innych **niejawnych** zależności (np. od konkretnego frameworku utrwalania danych), a prezentowane do tej pory podejście wymagałoby kodowania tych zależności na stałe w kodzie źródłowym programu.

Spójrzmy teraz na prosty przykład tego, jak kontener Springa może obsługiwać mechanizm odwracania kontroli. W pierwszej kolejności przebudujemy naszą usługę pogodową, aby prezentowała właściwy podział na interfejs i implementację oraz umożliwiała definiowanie (konfigurowanie) dla tej implementacji konkretnego egzemplarza obiektu DAO w formie właściwości komponentu JavaBean:

```
public interface WeatherService {
   Double getHistoricalHigh(Date date);
}
```

```
public class WeatherServiceImpl implements WeatherService {

  private WeatherDao weatherDao;

  public void setWeatherDao(WeatherDao weatherDao) {
    this.weatherDao = weatherDao;
  }

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

// ta sama klasa co w poprzednim przykładzie
public class StaticDataWeatherDaoImpl implements WeatherDao {
    ...
}
```

Do zarządzania egzemplarzem usługi pogodowej użyjemy kontekstu aplikacji Springa, a konkretnie klasy `ClassPathXmlApplicationContext`, i upewnimy się, że kontekst aplikacji otrzymał obiekt DAO naszej usługi, z którym będzie mógł współpracować. W pierwszej kolejności musimy zdefiniować plik konfiguracyjny w formacie XML, *applicationContext.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="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref local="weatherDao"/>
```

```
    </property>
  </bean>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

Obiekt `weatherService` konfigurujemy za pomocą elementu `bean`, w którym określamy, że jego właściwość, `weatherDao`, powinna być ustawiona na egzemplarz komponentu `weatherDao` (który także definiujemy jako element `bean`). Pozostaje nam już tylko zmodyfikowanie klasy testowej, aby tworzyła odpowiedni kontener i uzyskiwała dostęp do znajdującej się w tym kontenerze usługi pogodowej:

```
public class WeatherServiceTest extends TestCase {
  public void testSample2() throws Exception {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
        "ch02/sample2/applicationContext.xml");
    WeatherService ws = (WeatherService)ctx.getBean("weatherService");

    Double high = ws.getHistoricalHigh(new GregorianCalendar(2004, 0, 1).getTime());
    // ... w tym miejscu powinieneś umieścić dodatkowy kod sprawdzający zwracaną wartość...
  }
}
```

Wszystkie analizowane klasy zostały już zakodowane i wdrożone zgodnie z zaleceniami IoC. Usługa pogodowa nie ma i nie potrzebuje żadnej wiedzy o szczegółach implementacji wykorzystywanego obiektu DAO, ponieważ zależność pomiędzy tymi składnikami aplikacji bazuje wyłącznie na interfejsie `WeatherDao`. Co więcej, podział oryginalnej klasy `Weather-➥Service` na interfejs `WeatherService` i implementującą go klasę `WeatherServiceImpl` chroni klientów usługi pogodowej przed takimi szczegółami implementacyjnymi jak sposób lokalizowania przez tę usługę odpowiedniego obiektu DAO (w tym konkretnym przypadku wykorzystano metodę ustawiającą znaną z komponentów JavaBeans). Takie rozwiązanie powoduje, że teraz implementacja usługi pogodowej może być zmieniana w sposób przezroczysty. Okazuje się, że na tym etapie możemy wymieniać implementacje interfejsu `WeatherDao` i (lub) interfejsu `WeatherService`, i że tego rodzaju zmiany będą wymagały wyłącznie odpowiedniego dostosowania zapisów pliku konfiguracyjnego (bez najmniejszego wpływu na funkcjonowanie klientów zmienianych komponentów). Przedstawiony przykład dobrze ilustruje zastosowanie interfejsów w sytuacji, gdy kod jednej z warstw wykorzystuje kod innej warstwy (wspominano o tym w rozdziale 1.).

Różne formy wstrzykiwania zależności

Termin **wstrzykiwanie zależności** (ang. *Dependency Injection*) opisuje proces dostarczania komponentowi niezbędnych zależności z wykorzystaniem techniki IoC — można więc powiedzieć, że zależności są w istocie wstrzykiwane do komponentów. Wersja wstrzykiwania zależności, którą mieliśmy okazję analizować w poprzednim przykładzie, jest nazywana **wstrzykiwaniem przez metody ustawiające** (ang. *Setter Injection*), ponieważ do wprowadzania (wstrzykiwania) zależności do właściwych obiektów wykorzystuje się znane z komponentów JavaBeans metody ustawiające. Więcej informacji na temat komponentów JavaBeans oraz specyfikację tej technologii znajdziesz na stronie internetowej *http://java.sun.➥com/products/javabeans*.

Przeanalizujmy teraz nieco inny wariant wstrzykiwania zależności — **wstrzykiwanie przez konstruktor** (ang. *Constructor Injection*), gdzie zależności są dostarczane do obiektu za pośrednictwem jego własnego konstruktora:

```
public interface WeatherService {
   Double getHistoricalHigh(Date date);
}
public class WeatherServiceImpl implements WeatherService {

  private final WeatherDao weatherDao;

  public WeatherServiceImpl(WeatherDao weatherDao) {
    this.weatherDao = weatherDao;
  }

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = weatherDao.find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

// niezmieniony interfejs WeatherDao
public interface WeatherDao {
  ...
}

// niezmieniona klasa StaticDataWeatherDaoImpl
public class StaticDataWeatherDaoImpl implements WeatherDao {
  ...
}
```

Klasa `WeatherServiceImpl` zawiera teraz konstruktor, który pobiera w formie argumentu egzemplarz interfejsu `WeatherDao` (zamiast — jak w poprzednim przykładzie — odpowiedniej metody ustawiającej, która także pobierała na wejściu egzemplarz tego interfejsu). Oczywiście także konfiguracja kontekstu aplikacji wymaga odpowiedniej modyfikacji. Okazuje się jednak, że zmiany w żaden sposób nie dotyczą klasy testowej, która nie wymaga żadnych dodatkowych czynności:

```
<beans>
  <bean id="weatherService" class="ch02.sample3.WeatherServiceImpl">
    <constructor-arg>
      <ref local="weatherDao"/>
    </constructor-arg>
  </bean>
  <bean id="weatherDao" class="ch02.sample3.StaticDataWeatherDaoImpl">
  </bean>
</beans>

// niezmieniona klasa WeatherServiceTest
public class WeatherServiceTest extends TestCase {
  ...
}
```

Wstrzykiwanie metod jest ostatnią formą wstrzykiwania zależności, którą się zajmiemy (jest stosowana zdecydowanie rzadziej od dwóch poprzednich technik). Wstrzykiwanie zależności w tej formie przenosi na kontener odpowiedzialność za implementowanie metod w czasie wykonywania programu. Przykładowo, obiekt może definiować chronioną metodę abstrakcyjną, a kontener może tę metodę implementować w czasie wykonywania programu w taki sposób, aby zwracała obiekt zlokalizowany podczas przeszukiwania środowiska wykonywania aplikacji. Celem wstrzykiwania metod jest oczywiście wyeliminowanie zależności od interfejsów API kontenera i tym samym ograniczanie niepożądanych związków.

Jednym z podstawowych i jednocześnie najlepszych zastosowań techniki wstrzykiwania metod jest obsługa przypadków, w których bezstanowy obiekt, singleton, musi współpracować z niesingletonem, z obiektem stanowym lub takim, który nie zapewnia bezpieczeństwa przetwarzania wielowątkowego. Przypomnij sobie naszą usługę pogodową, w której potrzebujemy tylko jednego egzemplarza, ponieważ nasza implementacja jest bezstanowa i jako taka może być wdrażana za pomocą domyślnego kontenera Springa, który traktuje ją jak singleton (tworzy tylko jeden egzemplarz, który można przechowywać w pamięci podręcznej i wykorzystywać wielokrotnie). Warto się jednak zastanowić, co by było, gdyby usługa pogodowa musiała korzystać z klasy `StatefulWeatherDao`, implementacji interfejsu `WeatherDao` niezapewniającej bezpieczeństwa wątków. W każdym wywołaniu metody `WeatherService.getHistoricalHigh()` usługa pogodowa musiałaby wykorzystywać świeży egzemplarz obiektu DAO (lub przynajmniej musiałaby się upewniać, że w danej chwili żaden inny egzemplarz samej usługi nie wykorzystuje tego samego obiektu DAO). Jak się za chwilę przekonasz, można w prosty sposób zasygnalizować kontenerowi konieczność traktowania DAO jak obiektu, który nie jest singletonem — wówczas każde żądanie dotyczące tego obiektu spowoduje zwrócenie nowego egzemplarza. Problem w tym, że pojedynczy egzemplarz usługi pogodowej jest przedmiotem wstrzykiwania zależności tylko raz. Jednym z rozwiązań jest oczywiście umieszczenie w kodzie tej usługi odwołań do interfejsu API kontenera Springa i — w razie potrzeby — żądanie nowego obiektu DAO. Takie podejście ma jednak jedną zasadniczą wadę: wiąże naszą usługę ze Springiem, co jest sprzeczne z dążeniem do jak największej izolacji obu struktur. Zamiast tego możemy zastosować oferowany przez Springa mechanizm **wstrzykiwania metody wyszukującej** (ang. *Lookup Method Injection*), gdzie usługa pogodowa będzie miała dostęp do odpowiedniego obiektu DAO za pośrednictwem metody JavaBean `getWeatherDao()`, która — w zależności od potrzeb — może być albo konkretną implementacją albo metodą abstrakcyjną. Następnie, już w definicji fabryki, nakazujemy kontenerowi przykrycie tej metody i zapewnienie implementacji zwracającej nowy egzemplarz DAO w formie innego komponentu:

```
public abstract class WeatherServiceImpl implements WeatherService {

  protected abstract WeatherDao getWeatherDao();

  public Double getHistoricalHigh(Date date) {
    WeatherData wd = getWeatherDao().find(date);
    if (wd != null)
      return new Double(wd.getHigh());
    return null;
  }
}

<beans>
  <bean id="weatherService" class="ch02.sample4.WeatherServiceImpl">
```

```
    <lookup-method name="getWeatherDao" bean="weatherDao"/>
  </bean>
  <bean id="weatherDao" singleton="false"
        class="ch02.sample4.StatefulDataWeatherDaoImpl">
  </bean>
</beans>
```

Zauważ, że nakazaliśmy kontenerowi, aby nie traktował obiektu DAO jak singleton, zatem każde wywołanie metody `getWeatherDao()` może zwrócić nowy egzemplarz tego obiektu. Gdybyśmy tego nie zrobili, metoda za każdym razem zwracałaby ten sam egzemplarz singletonu (składowany w pamięci podręcznej). Takie rozwiązanie jest co prawda poprawne, jednak jest mało prawdopodobne, byś kiedykolwiek chciał uzyskać właśnie taki efekt, ponieważ podstawową zaletą techniki wstrzykiwania metod wyszukujących jest możliwość wstrzykiwania prototypów (niesingletonów), jak choćby w analizowanym przykładzie. W tym przypadku klasa `WeatherServiceImpl` i metoda `getWeatherDao()` są abstrakcyjne, możemy jednak przykryć dowolną metodę zwracającą każdego komponentu JavaBean (w praktyce metody kandydujące nie muszą pobierać żadnych argumentów, a ich nazwy nie muszą nawet być zgodne z konwencją nazewnictwa JavaBeans, choć ze względu na klarowność kodu takie rozwiązanie jest zalecane). Warto pamiętać, że także pozostały kod aplikacji musi korzystać z naszej metody zwracającej (a nie z odpowiedniego pola) do uzyskiwania dostępu do obiektu DAO. Wywoływanie metod zamiast bezpośredniego odwoływania się do pól obiektów jest jeszcze jedną dobrą praktyką przetwarzania właściwości komponentów JavaBeans.

Stosując techniki wstrzykiwania metod, warto odpowiedzieć sobie na pytanie, jak można testować kod (np. wykonywać testy jednostkowe) aplikacji bez kontenera, który wstrzykuje daną metodę. Podobne wątpliwości są szczególnie istotne w przypadku aplikacji podobnych do analizowanej usługi pogodowej, gdzie klasa implementacji jest abstrakcyjna. Stosunkowo prostą i jednocześnie możliwą do zrealizowania strategią przeprowadzania testów jednostkowych w podobnych przypadkach jest stworzenie dla wspomnianej klasy abstrakcyjnej podklasy zawierającej testową implementację metody, która w normalnych warunkach jest wstrzykiwana, przykładowo:

```
...
WeatherService ws = new WeatherServiceImpl() {
  protected WeatherDao getWeatherDao() {
    // zwraca obiekt DAO dla danego testu
    ...
  }
};
```

Stosowanie przez użytkowników Springa tak zaawansowanych mechanizmów jak wstrzykiwanie metod zawsze powinno być poprzedzone dogłębną analizą — należy zdecydować, które rozwiązanie w danej sytuacji jest najbardziej uzasadnione i które nie będzie stanowiło niepotrzebnego utrudnienia dla programistów. Niezależnie od wyników tej analizy naszym zdaniem przedstawione powyżej podejście jest pod wieloma względami lepsze od umieszczania w kodzie aplikacji zależności od interfejsów API kontenera.

Wybór pomiędzy wstrzykiwaniem przez metody ustawiające a wstrzykiwaniem przez konstruktory

Podczas korzystania z istniejących klas może się zdarzyć, że nie będziesz nawet miał wyboru pomiędzy użyciem techniki wstrzykiwania przez metody ustawiające a zastosowaniem wstrzykiwania przez konstruktor. Jeśli dana klasa zawiera wyłącznie wieloargumentowy konstruktor i żadnych właściwości komponentu JavaBean lub jednoargumentowy konstruktor i proste właściwości JavaBean, wybór metody wstrzykiwania nie zależy od Ciebie — dokonano go już wcześniej. Co więcej, niektóre spośród istniejących klas mogą wymuszać stosowanie kombinacji obu form wstrzykiwania zależności, gdzie oprócz wieloargumentowego konstruktora będziesz zobligowany do ustawienia kilku opcjonalnych właściwości JavaBean.

Jeśli masz wybór odnośnie wersji wstrzykiwania zależności, której chcesz użyć lub dla której chcesz zaprojektować architekturę swojej aplikacji, przed podjęciem ostatecznej decyzji powinieneś wziąć pod uwagę kilka czynników:

- Stosowanie właściwości JavaBean generalnie ułatwia obsługę domyślnych lub opcjonalnych wartości w sytuacji, gdy nie wszystkie wartości faktycznie są wymagane. W przypadku wstrzykiwania przez konstruktor takie podejście musiałoby prowadzić do stworzenia wielu wariantów konstruktora, gdzie jeden konstruktor wywoływałby w swoim ciele inny. Wiele wersji konstruktora i długie listy argumentów mogą być zbyt rozwlekłe i utrudniać konserwację kodu.
- W przeciwieństwie do konstruktorów właściwości JavaBean (jeśli nie są prywatne) są automatycznie dziedziczone przez podklasy. Ograniczenie związane z dziedziczeniem konstruktorów często zmusza programistów do tworzenia w kodzie podklas szablonowych konstruktorów, których jedynym zadaniem jest wywoływanie odpowiednich konstruktorów nadklasy. Warto jednak pamiętać, że większość środowisk IDE oferuje obecnie rozmaite mechanizmy uzupełniania kodu, które czynią proces tworzenia konstruktorów lub właściwości JavaBean niezwykle łatwym.
- Właściwości JavaBean są zdecydowanie łatwiejsze w interpretacji (nawet bez stosowania dodatkowych komentarzy dokumentujących) na poziomie kodu źródłowego niż argumenty konstruktora. Właściwości JavaBean wymagają też mniejszego wysiłku podczas dodawania komentarzy JavaDoc, ponieważ (w przeciwieństwie do konstruktorów) nie powtarzają się w kodzie.
- W czasie działania, właściwości JavaBean mogą być dopasowywane według nazw, które są widoczne z poziomu mechanizmu refleksji Javy. Z drugiej strony, w skompilowanym pliku klasy użyte w kodzie źródłowym nazwy argumentów konstruktora nie są zachowywane, zatem automatyczne dopasowywanie według nazw jest niemożliwe.
- Właściwości JavaBean oferują możliwość zwracania (i ustawiania) bieżącego stanu, jeśli tylko zdefiniowano odpowiednią metodę zwracającą (i ustawiającą). Takie rozwiązanie jest w wielu sytuacjach niezwykle przydatne, np. kiedy należy zapisać bieżący stan w innymi miejscu.

- Interfejs `PropertyEditor` dla właściwości JavaBean umożliwia wykonywanie automatycznej konwersji typów tam, gdzie jest to konieczne. Właśnie to wygodne rozwiązanie jest wykorzystywane i obsługiwane przez Springa.
- Właściwości JavaBean mogą być zmienne, ponieważ odpowiednie metody ustawiające można wywoływać wiele razy. Oznacza to, że jeśli wymaga tego realizowany przypadek testowy, można w prosty sposób modyfikować istniejące zależności. Inaczej jest w przypadku zależności przekazywanych za pośrednictwem konstruktorów, które — jeśli nie są dodatkowo udostępniane w formie właściwości — pozostają niezmienne. Z drugiej strony, jeśli jednym z wymagań jest bezwzględna niezmienność zależności, właśnie technika wstrzykiwania przez konstruktor umożliwia zadeklarowanie pola ustawianego w ciele konstruktora ze słowem kluczowym `final` (metoda ustawiająca w najlepszym razie może odpowiedzieć wyjątkiem na próbę drugiego lub kolejnego wywołania; nie ma też możliwości zadeklarowania pola ustawianego w takiej metodzie ze słowem kluczowym `final`, które zabezpieczyłoby to pole przed ewentualnymi modyfikacjami przez pozostały kod danej klasy).
- Argumenty konstruktora dają programiście pewność, że poprawny obiekt zostanie skonstruowany, ponieważ zadeklarowanie wszystkich wymaganych wartości wymusi ich przekazanie, a sama klasa w procesie inicjalizacji może je wykorzystywać w wymaganej kolejności w procesie inicjalizacji. W przypadku właściwości JavaBean nie można wykluczyć sytuacji, w której część właściwości nie została ustawiona przed użyciem danego obiektu, który — tym samym — znajdował się w niewłaściwym stanie. Co więcej, jeśli użyjesz właściwości JavaBean, nie będziesz mógł w żaden sposób wymusić kolejności wywołań metod ustawiających, co może wymagać dodatkowych zabiegów inicjalizujących już po ustawieniu właściwości (za pomocą metody `init()`). Taka dodatkowa metoda może też sprawdzać, czy wszystkie właściwości zostały ustawione. Spring oferuje mechanizmy automatycznego wywoływania dowolnej zadeklarowanej metody inicjalizującej bezpośrednio po ustawieniu wszystkich właściwości; alternatywnym rozwiązaniem jest zaimplementowanie interfejsu `InitializingBean`, który deklaruje automatycznie wywoływaną metodę `afterPropertiesSet()`.
- Stosowanie kilku konstruktorów może być bardziej zwięzłym rozwiązaniem niż korzystanie z wielu właściwości JavaBean wraz z odpowiednimi metodami ustawiającymi i zwracającymi. Większość środowisk IDE oferuje jednak funkcje uzupełniania kodu, które bardzo ułatwiają i skracają proces tworzenia zarówno konstruktorów, jak i właściwości JavaBean.

Ogólnie, zespół Springa w większości praktycznych zastosowań preferuje stosowanie techniki wstrzykiwania przez metody ustawiające zamiast wstrzykiwania przez konstruktory, choć tego rodzaju decyzje powinny być dobrze przemyślane i w żadnym razie nie należy uprawiać doktrynerstwa. Wymienione powyżej aspekty obejmują większość czynników, które należy brać pod uwagę w każdej z sytuacji. Przyjmuje się, że wstrzykiwanie przez konstruktory lepiej się sprawdza w przypadku prostszych scenariuszy inicjalizacji, gdzie konstruktor otrzymuje na wejściu zaledwie kilka argumentów, które dodatkowo powinny reprezentować złożone, a więc łatwiejsze do dopasowania, i unikatowe typy. Wraz ze wzrostem złożoności całej konfiguracji rośnie przewaga techniki wstrzykiwania przez metody ustawiające (zarówno w zakresie możliwości konserwacji, jak i nakładów pracy samego programisty). Warto pamiętać, że także inne popularne kontenery IoC obsługują zarówno

wstrzykiwanie przez konstruktory, jak i wstrzykiwanie przez metody ustawiające, zatem obawa przed nadmiernym związaniem aplikacji z mechanizmami Springa nie powinna być uwzględniana podczas dokonywania tego wyboru.

Niezależnie od wybranej formy wstrzykiwania zależności jest oczywiste, że należy ten mechanizm stosować w taki sposób, aby odpowiednie konstrukcje były wprowadzane wyłącznie do klasy implementacji oraz właściwej konfiguracji tej klasy. Pozostały kod aplikacji powinien współpracować z pozostałymi modułami (komponentami) za pośrednictwem interfejsów i w żaden sposób nie powinien być uzależniony od problemów konfiguracyjnych.

Kontener

Podstawowy kontener IoC Springa jest często nazywany fabryką komponentów (ang. *bean factory*). Każda fabryka komponentów umożliwia logicznie spójne konfigurowanie i wiązanie wielu obiektów za pomocą odpowiednich mechanizmów wstrzykiwania zależności. Fabryka komponentów oferuje też pewne funkcje w zakresie zarządzania obiektami, a w szczególności ich cyklem życia. Podejście bazujące na technice odwracania kontroli umożliwia daleko idące oddzielanie kodu poszczególnych składników aplikacji. Co więcej, poza korzystaniem z mechanizmu refleksji podczas uzyskiwania dostępu do obiektów odwracanie kontroli uniezależnia kod aplikacji od samej fabryki komponentów i — tym samym — eliminuje konieczność dostosowywania kodu do współpracy z funkcjami Springa. Kod aplikacji potrzebny do konfigurowania obiektów (i do uzyskiwania obiektów z wykorzystaniem takich wzorców jak singletony czy fabryki obiektów specjalnych) można albo w całości wyeliminować, albo przynajmniej w znacznym stopniu zredukować.

> W typowej aplikacji bazującej na Springu tylko bardzo niewielka ilość kodu **scalającego** będzie świadoma do współpracy z kontenerem Springa lub korzystania z interfejsów tego kontenera. Nawet tę skromną ilość kodu w wielu przypadkach udaje się wyeliminować, korzystając z istniejącego kodu frameworka do załadowania fabryki komponentów w sposób deklaratywny.

Fabryka komponentów

Istnieją różne implementacje fabryki komponentów, z których każda oferuje nieco inną funkcjonalność (w praktyce różnice mają związek z działaniem mechanizmów konfiguracji, a najczęściej stosowaną reprezentacją jest format XML). Wszystkie fabryki komponentów implementują interfejs `org.springframework.beans.factory.BeanFactory` — programowe zarządzanie egzemplarzami wymaga dostępności właśnie za pośrednictwem tego interfejsu. Dodatkową funkcjonalność udostępniają ewentualne podinterfejsy interfejsu `BeanFactory`. Poniższa lista zawiera część tej hierarchii interfejsów:

- `BeanFactory` — podstawowy interfejs wykorzystywany do uzyskiwania dostępu do wszystkich fabryk komponentów. Przykładowo wywołanie metody `getBean(String name)` umożliwia uzyskanie komponentu z kontenera na podstawie nazwy. Inny wariant tej metody, `getBean(String name, Class requiredType)`, dodatkowo daje

programiście możliwość określenia wymaganej klasy zwracanego komponentu i generuje wyjątek, jeśli taki komponent nie istnieje. Inne metody umożliwiają nam odpytywanie fabryki komponentów w zakresie istnienia poszczególnych komponentów (według nazw), odnajdywanie typów komponentów (także według nazw) oraz sprawdzanie, czy w danej fabryce istnieją jakieś aliasy danego komponentu (czy dany komponent występuje w fabryce pod wieloma nazwami). I wreszcie można określić, czy dany komponent skonfigurowano jako singleton (czy pierwszy utworzony egzemplarz komponentu będzie wykorzystywany wielokrotnie dla wszystkich kolejnych żądań czy fabryka za każdym razem będzie tworzyła nowy egzemplarz).

- `HierarchicalBeanFactory` — większość fabryk komponentów może być tworzona jako część większej hierarchii — wówczas żądanie od fabryki dostępu do komponentu, który w tej konkretnej fabryce nie istnieje, spowoduje przekazanie żądania do jego fabryki nadrzędnej (przodka), która może z kolei zażądać odpowiedniego komponentu od swojej fabryki nadrzędnej itd. Z perspektywy aplikacji klienta cała hierarchia powyżej bieżącej fabryki (włącznie z tą fabryką) może być traktowana jak jedna, scalona fabryka. Jedną z zalet takiej hierarchicznej struktury jest możliwość jej dopasowania do rzeczywistych warstw architekturalnych lub modułów aplikacji. O ile uzyskiwanie komponentu z fabryki będącej częścią hierarchii odbywa się w sposób całkowicie przezroczysty, interfejs `HierarchicalBeanFactory` jest niezbędny, jeśli chcemy mieć możliwość dostępu do komponentów składowanych w jej fabryce nadrzędnej.

- `ListableBeanFactory` — metody tego podinterfejsu interfejsu `BeanFactory` umożliwiają generowanie rozmaitych list komponentów dostępnych w bieżącej fabryce, włącznie z listą nazw komponentów, nazwami wszystkich komponentów określonego typu oraz liczbą komponentów w danej fabryce. Kilka metod tego interfejsu umożliwia dodatkowo tworzenie egzemplarza `Map` zawierającego wszystkie komponenty określonego typu. Warto pamiętać, że o ile metody interfejsu `BeanFactory` automatycznie są udostępniane na wszystkich poziomach hierarchii fabryk komponentów, metody interfejsu `ListableBeanFactory` są stosowane tylko dla jednej, bieżącej fabryki. Klasa pomocnicza `BeanFactoryUtils` oferuje niemal identyczne metody jak interfejs `ListableBeanFactory`, tyle że w przeciwieństwie do metod tego interfejsu uwzględniają w generowanych wynikach całą hierarchię fabryk komponentów. W wielu przypadkach stosowanie metod klasy `BeanFactoryUtils` jest lepszym rozwiązaniem niż korzystanie z interfejsu `ListableBeanFactory`.

- `AutowireCapableBeanFactory` — interfejs `AutowireCapableBeanFactory` umożliwia (za pośrednictwem metod `autowireBeanProperties()` oraz `applyBeanPropertyValues()`) konfigurowanie istniejącego, zewnętrznego obiektu i dostarczanie zależności z poziomu fabryki komponentów. Oznacza to, że wspomniane metody pozwalają pominąć normalny krok tworzenia obiektu będący częścią uzyskiwania komponentu za pośrednictwem metody `BeanFactory.getBean()`. Podczas pracy z zewnętrznym kodem, który wymaga tworzenia egzemplarzy obiektów jako takich, tworzenie komponentów za pomocą odpowiednich mechanizmów Springa nie zawsze jest możliwe, co nie oznacza, że należy rezygnować z oferowanych przez ten framework cennych rozwiązań w zakresie wstrzykiwania zależności. Inna metoda, `autowire()`, umożliwia określanie na potrzeby fabryki komponentów nazwy klasy, wykorzystanie tej fabryki do utworzenia egzemplarza tej klasy, użycie refleksji do wykrycia wszystkich zależności tej klasy i wstrzyknięcia tych zależności do danego

komponentu (w efekcie powinieneś otrzymać w pełni skonfigurowany obiekt). Warto pamiętać, że fabryka nie będzie zawierała i zarządzała obiektami konfigurowanymi za pomocą tych metod (inaczej niż w przypadku tworzonych przez siebie egzemplarzy singletonów), choć możemy sami dodać do tej fabryki nasze egzemplarze jako singletony.

- `ConfigurableBeanFactory` — ten interfejs wprowadza do podstawowej fabryki komponentów dodatkowe opcje konfiguracyjne, które mogą być stosowane w fazie inicjalizacji.

Ogólnie Spring próbuje stosować tzw. wyjątki nieweryfikowalne (ang. *non-checked exceptions*), czyli podklasy klasy `RuntimeException` dla nieweryfikowalnych błędów. Interfejsy fabryki komponentów, w tym `BeanFactory` i jego podinterfejsy, nie są pod tym względem wyjątkiem. W większości przypadków błędy konfiguracji są nieweryfikowalne, zatem wszystkie wyjątki zwracane przez opisywane interfejsy API są podklasami klasy nieweryfikowalnych wyjątków `BeansException`. Decyzja o tym, gdzie i kiedy należy przechwytywać i obsługiwać te wyjątki, należy do programisty — jeśli istnieje odpowiednia strategia obsługi sytuacji wyjątkowej, można nawet podejmować próby odtwarzania stanu sprzed wyjątku.

Kontekst aplikacji

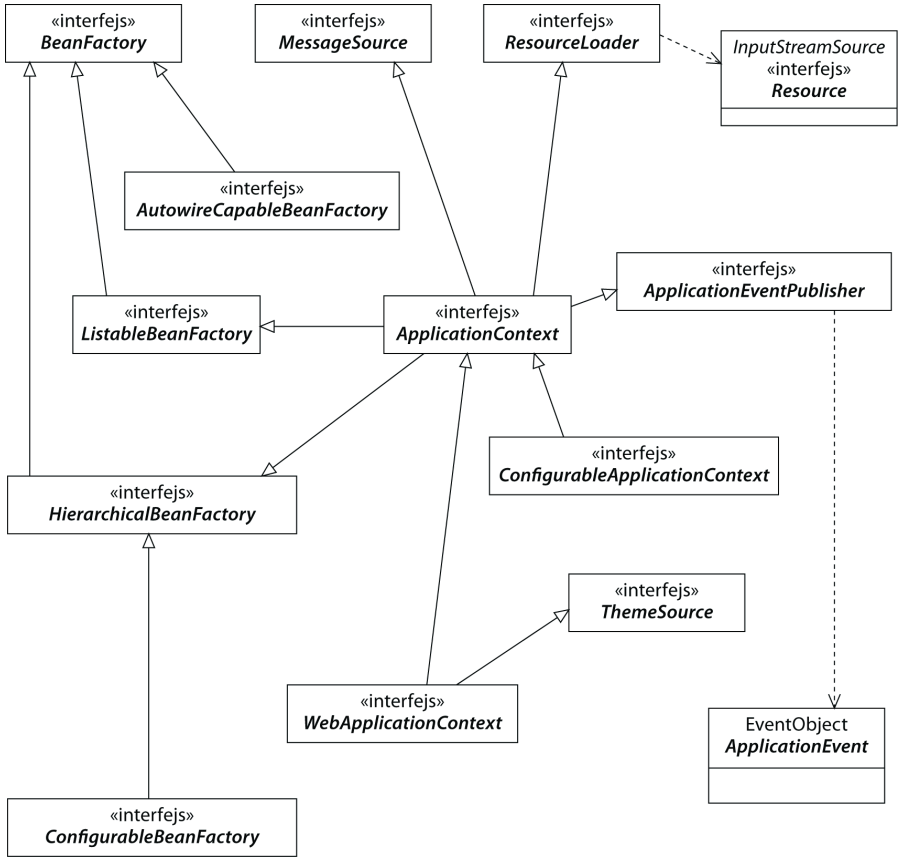
Spring obsługuje też mechanizm znacznie bardziej zaawansowany od fabryki komponentów — tzw. **kontekst aplikacji** (ang. *application context*).

> Warto podkreślić, że kontekst aplikacji jest fabryką komponentów, a interfejs `org.sp-➥ringframework.context.ApplicationContext` jest podinterfejsem interfejsu `BeanFactory`. Kompletną hierarchię interfejsów przedstawiono na rysunku 2.1.

Ogólnie wszystkie zadania, które można realizować za pomocą fabryki komponentów, można też wykonywać z wykorzystaniem kontekstu aplikacji. Po co więc wprowadzono takie rozróżnienie? Przede wszystkim po to, aby podkreślić zwiększoną funkcjonalność i nieco inny sposób używania kontekstu aplikacji:

- **Ogólny styl pracy *z frameworkiem*** — pewne operacje na komponentach w kontenerze lub na samym kontenerze, które w fabryce komponentów muszą być obsługiwane programowo, w kontekście aplikacji mogą być realizowane deklaratywnie. Chodzi między innymi o rozpoznawanie i stosowanie specjalnych postprocesorów dla komponentów oraz postprocesorów dla fabryki komponentów. Co więcej, istnieje wiele mechanizmów Springa, które ułatwiają automatyczne wczytywanie kontekstów aplikacji — przykładowo, w warstwie MVC aplikacji internetowych większość fabryk komponentów będzie tworzona przez kod użytkownika, natomiast konteksty aplikacji będą wykorzystywane najczęściej za pomocą technik deklaratywnych i tworzone przez kod Springa. W obu przypadkach znaczna część kodu użytkownika będzie oczywiście zarządzana **przez** kontener, zatem nie będzie dysponowała żadną wiedzą **o** samym kontenerze.
- **Obsługa interfejsu** `MessageSource` — kontekst aplikacji implementuje interfejs `MessageSource`, którego zadaniem jest uzyskiwanie zlokalizowanych komunikatów i którego implementacja może być w prosty sposób włączana do aplikacji.

Rysunek 2.1.



- **Obsługa zdarzeń aplikacji i frameworka** — kontekst aplikacji oferuje możliwość wywoływania zdarzeń frameworka lub aplikacji, które są następnie przechwytywane i obsługiwane przez zarejestrowane obiekty nasłuchujące.
- **Obsługa interfejsu** `ResourceLoader` — interfejs `ResourceLoader` Springa jest elastyczną i uniwersalną abstrakcją obsługującą niskopoziomowe zasoby. Sam kontekst aplikacji jest implementacją tego interfejsu i — tym samym — zapewnia aplikacji dostęp do egzemplarzy `Resource` właściwych dla danego wdrożenia.

Być może zastanawiasz się, kiedy lepszym rozwiązaniem jest tworzenie i korzystanie z fabryki komponentów, a kiedy lepiej użyć kontekstu aplikacji. Niemal we wszystkich przypadkach warto rozważyć użycie kontekstu aplikacji, który bez żadnych dodatkowych kosztów poszerza zakres funkcjonalności. Prawdopodobnie jedynym wyjątkiem od tej reguły są aplikacje bazujące na apletach, gdzie każdy zajmowany (lub przesyłany) bajt pamięci może mieć ogromne znaczenie — fabryka komponentów pozwala na pewne oszczędności w tym zakresie, ponieważ umożliwia korzystanie z pakietu bibliotek Springa, które zawierają wyłącznie funkcjonalność fabryki (bez rozbudowanej funkcjonalności kontekstu aplikacji). W tym i innych rozdziałach poświęconych funkcjonalności fabryki komponentów możesz bezpiecznie zakładać, że wszystkie analizowane elementy funkcjonalności są wspólne zarówno dla fabryki komponentów, jak i dla kontekstu aplikacji.

W tym i w większości pozostałych rozdziałów tej książki analiza konfiguracji i funkcjonalności kontenera będzie ilustrowana przykładami w wersji deklaratywnej, opartej na formacie XML (w tym `XmlBeanFactory` lub `ClassPathXmlApplicationContext`) fabryki komponentów i kontekstu aplikacji. Warto zdać sobie sprawę z tego, że funkcjonalność kontenera w żadnym razie nie jest tożsama z formatem jego konfiguracji. Konfiguracja oparta na XML jest co prawda wykorzystywana przez zdecydowaną większość użytkowników Springa, jednak istnieje też kompletny interfejs API dla mechanizmów konfiguracji i dostępu do kontenerów, w tym obsługa innych formatów konfiguracji (które mogą być konstruowane i obsługiwane w bardzo podobny sposób jak odpowiednie dokumenty XML). Przykładowo, istnieje klasa `PropertiesBeanDefinitionReader`, która odpowiada za załadowanie do fabryki komponentów definicji zapisanych w plikach właściwości Javy.

Uruchamianie kontenera

Poprzednie przykłady pokazały, jak można programowo uruchamiać kontener z poziomu kodu użytkownika. W tym punkcie przeanalizujemy kilka ciekawych rozwiązań w tym zakresie.

Egzemplarz (implementację) interfejsu `ApplicationContext` można wczytać, wskazując ścieżkę do odpowiedniej klasy (pliku):

```
ApplicationContext appContext =
    new ClassPathXmlApplicationContext("ch03/sample2/applicationContext.xml");
// Uwaga: ApplicationContext jest egzemplarzem BeanFactory, to oczywiste!
BeanFactory factory = (BeanFactory) appContext;
```

Można też wskazać konkretną lokalizację w systemie plików:

```
ApplicationContext appContext =
    new FileSystemXmlApplicationContext("/some/file/path/applicationContext.xml");
```

Istnieje też możliwość łączenia dwóch lub większej liczby fragmentów prezentowanych już dokumentów XML. Takie rozwiązanie umożliwia nam lokalizowanie definicji komponentów w ramach logicznych modułów, do których należą, i jednocześnie tworzenie jednego kontekstu na bazie połączonych definicji. Przykład prezentowany w następnym rozdziale pokaże, że opisane podejście może być bardzo użyteczne także podczas testów. Poniżej przedstawiono jeden z wcześniejszych przykładów konfiguracji, tyle że tym razem złożony z dwóch fragmentów kodu XML:

applicationContext-dao.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="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

applicationContext-services.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="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref bean="weatherDao"/>
    </property>
  </bean>
</beans>
```

Uważni Czytelnicy zapewne zauważyli, że w porównaniu z wcześniejszymi przykładami nieznacznie zmieniono sposób odwoływania się do komponentu `weatherDao`*, który wykorzystujemy jako właściwość komponentu usługi pogodowej; odwołania do komponentów szczegółowo omówimy w dalszej części rozdziału.*

Aby wczytać i połączyć oba fragmenty (teoretycznie może ich być więcej), musimy je tylko wymienić w tablicy nazw (łańcuchów) przekazywanej do konstruktora klasy `ClassPathXml-➥ApplicationContext`:

```
ApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext-services.xml",
                      "applicationContext-dao.xml"});
```

Abstrakcja `Resource` Springa (którą szczegółowo omówimy nieco później) umożliwia stosowanie prefiksu `classpath*:` do wyszczególniania tych **wszystkich** zasobów pasujących do konkretnej nazwy, które są widoczne z perspektywy classloadera i jego classloaderów nadrzędnych. Przykładowo, gdyby nasza aplikacja została rozproszona pomiędzy wiele plików JAR, wszystkich należących do ścieżki do klas, z których każdy zawierałby własny fragment kontekstu aplikacji nazwany *applicationContext.xml*, moglibyśmy w prosty sposób określić, że chcemy utworzyć kontekst złożony ze wszystkich istniejących fragmentów:

```
ApplicationContext appContext =
        new ClassPathXmlApplicationContext("classpath*:applicationContext.xml"});
```

Okazuje się, że tworzenie i wczytywanie fabryki komponentów skonfigurowanych za pomocą odpowiednich zapisów języka XML jest bardzo łatwe. Najprostszym rozwiązaniem jest użycie abstrakcji `Resource` Springa, która umożliwia uzyskiwanie dostępu do zasobów według ścieżki klas:

```
ClassPathResource res =
        new ClassPathResource("org/springframework/prospering/beans.xml"});
XmlBeanFactory factory = new XmlBeanFactory(res);
```

lub:

```
FilesystemResource res = new FilesystemResource("/some/file/path/beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

Równie dobrze moglibyśmy użyć egzemplarza `InputStream`:

```
InputStream is = new FileInputStream("/some/file/path/beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

Aby wyczerpać ten temat, musimy wspomnieć o możliwości prostego oddzielenia operacji tworzenia fabryki komponentów od procesu przetwarzania definicji komponentów. Nie będziemy się zajmować tym zagadnieniem w szczegółach, warto jednak pamiętać, że takie wyodrębnienie zachowania fabryki komponentów od mechanizmów przetwarzania definicji komponentów upraszcza stosowanie innych formatów konfiguracji:

```
ClassPathResource res = new ClassPathResource("beans.xml"});
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(res);
```

Stosując klasę `GenericApplicationContext` dla kontekstu aplikacji, możemy w podobny sposób oddzielić kod tworzący ten kontekst od kodu przetwarzającego definicje komponentów. Wiele aplikacji Springa w ogóle nie tworzy kontenera programowo, ponieważ bazuje na odpowiednim kodzie tego frameworka (który wykonuje odpowiednie działania w ich imieniu). Przykładowo, możemy deklaratywnie skonfigurować mechanizm `ContextLoader` Springa w taki sposób, aby automatycznie wczytywał kontekst aplikacji w momencie uruchamiania aplikacji internetowej. Odpowiednie techniki konfiguracji omówimy w następnym rozdziale.

Korzystanie komponentów uzyskiwanych z fabryki

Kiedy już fabryka komponentów lub kontekst aplikacji zostaną wczytane, uzyskanie dostępu do komponentów sprowadza się do wywołania metody `getBean()` interfejsu `BeanFactory`:

```
WeatherService ws = (WeatherService)ctx.getBean("weatherService");
```

lub jednej z metod któregoś z bardziej zaawansowanych interfejsów:

```
Map allWeatherServices = ctx.getBeansOfType(WeatherService.class);
```

Żądanie od kontenera dostępu do komponentu wywołuje proces jego tworzenia i inicjalizacji, który obejmuje omówioną już fazę wstrzykiwania zależności. Krok wstrzykiwania zależności może z kolei zainicjować proces tworzenia pozostałych komponentów (zależności pierwszego komponentu) itd., aż do utworzenia kompletnego grafu wzajemnie powiązanych egzemplarzy obiektów.

W związku z opisaną procedurą nasuwa się dość oczywiste pytanie: co należałoby zrobić z samą fabryką komponentów lub kontekstem aplikacji, aby pozostały kod, który tego wymaga, mógł uzyskać do nich dostęp? Obecnie skupiamy się na analizie sposobu konfigurowania i funkcjonowania kontenera, zatem przedstawienie i wyjaśnienie odpowiedzi na to pytanie odłóżmy na dalszą część tego rozdziału.

Pamiętaj, że z wyjątkiem bardzo niewielkiej ilości kodu scalającego zdecydowana większość kodu aplikacji pisanego i łączonego zgodnie z założeniami IoC nie musi zawierać żadnych operacji związanych z uzyskiwaniem dostępu do fabryki, ponieważ za zarządzanie zależnościami pomiędzy obiektami i samymi obiektami odpowiada kontener. Najprostszą strategią zapewniania odpowiednich proporcji pomiędzy kodem scalającym (niezbędnym do inicjowania pewnych działań) a właściwym kodem aplikacji jest umieszczenie fabryki komponentów w znanym miejscu, które będzie odpowiadało nie tylko oczekiwanym zastosowaniom, ale też tym elementom kodu, które będą potrzebowały dostępu do tej fabryki. Sam Spring oferuje mechanizm deklaratywnego wczytywania kontekstu dla aplikacji internetowych i składowania tego kontekstu w obiekcie `ServletContext`. Co więcej, Spring zawiera klasy pomocnicze przypominające singletony, które mogą być z powodzeniem stosowane do składowania i wydobywania z fabryki komponentów (jeśli takie rozwiązanie z jakiegoś powodu wydaje Ci się lepsze lub jeśli nie istnieje strategia składowania fabryki komponentów w ramach konkretnej aplikacji).

Konfiguracja komponentów w formacie XML

Mieliśmy już okazję przejrzeć przykładowe pliki z definicjami fabryk komponentów w formacie XML, jednak do tej pory nie poddaliśmy zawartych tam zapisów szczegółowej analizie. W największym uproszczeniu definicja fabryki komponentów składa się z elementu `beans` (na najwyższym poziomie) oraz jednego lub wielu elementów `bean`:

```
<?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="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref local="weatherDao"/>
    </property>
  </bean>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

Prawidłowe elementy i atrybuty pliku definicji szczegółowo opisano w pliku XML *DTD* (ang. *Document Type Definition*) nazwanym *spring-beans.dtd*. Wspomniany plik DTD w połączeniu z podręcznikiem użytkownika Springa powinien być traktowany jak ostateczne źródło informacji o technikach konfiguracji aplikacji. Ogólnie, opcjonalne atrybuty elementu `beans` (na najwyższym poziomie elementów XML definicji komponentów) mają wpływ na zachowanie całego pliku konfiguracji i zawierają domyślne wartości dla rozmaitych aspektów wszystkich definiowanych komponentów, natomiast większość opcjonalnych atrybutów i podelementów potomnych elementów `bean` opisuje konfigurację i cykl życia poszczególnych komponentów. Plik *spring-beans.dtd* jest co prawda dołączany do Springa, jednak z jego zawartością można się zapoznać także w internecie (patrz *www.springframework.org/dtd/spring-beans.dtd*).

Podstawowa definicja komponentu

Definicja pojedynczego komponentu zawiera wszystkie informacje, których kontener potrzebuje do jego utworzenia, w tym trochę szczegółowych danych o cyklu życia komponentu oraz jego zależnościach. Przyjrzyjmy się dwóm pierwszym składnikom elementu `bean`.

Identyfikator

W definicji komponentów na najwyższym poziomie hierarchii niemal we wszystkich przypadkach definiujemy jeden lub wiele identyfikatorów (lub nazw) komponentów, aby pozostałe komponenty mogły się do nich odwoływać podczas programowego korzystania z danego kontenera. Do definiowania identyfikatorów (lub głównych identyfikatorów) komponentów służy atrybut `id`. Zaletą tego atrybutu jest zgodność z typem XML IDREF — kiedy pozostałe komponenty będą się odwoływały do komponentu za pośrednictwem tak zdefiniowanego identyfikatora, sam parser dokumentu XML może pomóc w sprawdzeniu, czy takie odwołanie jest poprawne (czy w tym samym pliku zdefiniowano odpowiedni identyfikator), zatem atrybut `id` ułatwia wczesną weryfikację prawidłowości konfiguracji. Warto jednak pamiętać, że typ XML IDREF wprowadza pewne ograniczenia w kwestii akceptowanych znaków — identyfikatory muszą się rozpoczynać od litery, a na kolejnych pozycjach mogą zawierać dowolne znaki alfanumeryczne i znaki podkreślenia (ale nie mogą zawierać spacji). W większości przypadków opisane ograniczenia nie stanowią co prawda żadnego problemu, jeśli jednak z jakiegoś powodu chcesz je ominąć, możesz zdefiniować identyfikator w atrybucie `name`. Przykładowo, takie rozwiązanie jest uzasadnione, jeśli identyfikator komponentu jest poza kontrolą użytkownika i reprezentuje ścieżkę URL. Co więcej, atrybut `name` dopuszcza możliwość stosowania listy identyfikatorów oddzielonych przecinkami. Kiedy definicja komponentu definiuje więcej niż jeden identyfikator, np. w formie kombinacji atrybutu `id` i (lub) atrybutu `name`, wszystkie dodatkowe identyfikatory (poza pierwszym) należy traktować jak **aliasy**. Odwołania do wszystkich identyfikatorów są równie poprawne. Przeanalizujmy kilka przykładów:

```
<beans>

  <bean id="bean1" class="ch02.sample5.TestBean"/>

  <bean name="bean2" class="ch02.sample5.TestBean"/>

  <bean name="/myservlet/myaction" class="ch02.sample5.TestBean"/>

  <bean id="component1-dataSource"
        name="component2-dataSource,component3-dataSource"
        class="ch02.sample5.TestBean"/>

</beans>
```

Ponieważ trzeci definiowany komponent wymaga identyfikatora rozpoczynającego się od znaku ukośnika (`/`), użycie atrybutu `id` jest niemożliwe — musimy użyć atrybutu `name`. Zauważ, że czwarty komponent zawiera aż trzy identyfikatory, wszystkie równie poprawne. Być może zastanawiasz się, po co w ogóle miałbyś definiować więcej niż jeden identyfikator dla komponentu. Dobrą praktyką jest takie dzielenie konfiguracji według komponentów lub

modułów, aby dla każdego modułu istniał fragment pliku XML zawierający komponenty powiązane z tym modułem (wraz z ich zależnościami). Nazwy tych zależności mogą (jak w przedstawionym przykładzie) być definiowane z prefiksem właściwym dla danego komponentu (w tym przypadku `dataSource`). Kiedy fabryka komponentów lub kontekst aplikacji jest ostatecznie budowany z wielu przygotowanych wcześniej fragmentów (lub kiedy tworzona jest hierarchia kontekstów — patrz dalsza część tego rozdziału), każdy z tych komponentów będzie się odwoływał do tego samego, fizycznego bytu. Takie rozwiązanie należy traktować jak techniczną próbę rozwiązania problemu izolacji komponentów.

Mechanizm tworzenia komponentów

Może też zaistnieć konieczność wskazania kontenerowi sposobu, w jaki ten powinien tworzyć lub uzyskiwać egzemplarze komponentu, kiedy będzie tego potrzebował. Najczęściej stosowanym mechanizmem jest tworzenie komponentu za pośrednictwem jego konstruktora. Do określania nazwy klasy komponentu służy atrybut `class`. Za każdym razem, gdy kontener potrzebuje **nowego** egzemplarza komponentu, wewnętrznie wykonuje odpowiednik znanego z języka Java operatora `new`. Wszystkie prezentowane do tej pory przykłady wykorzystywały właśnie ten mechanizm.

Innym mechanizmem tworzenia komponentów jest sygnalizowanie kontenerowi konieczności użycia statycznej **metody fabrykującej** (ang. *factory method*), której jedynym zadaniem jest zwracanie nowego egzemplarza komponentu. Istniejący kod, nad którym nie masz kontroli, będzie Cię czasami zmuszał do korzystania z takiej statycznej metody fabrykującej. Nazwę klasy zawierającej tę metodę można określić za pośrednictwem atrybutu `class`, natomiast nazwę samej metody farykującej należy zdefiniować w atrybucie `factory-method`:

```
...
<bean id="testBeanObtainedViaStaticFactory"
      class="ch02.sample4.StaticFactory" factory-method="getTestBeanInstance"/>
...

public class StaticFactory {
  public static TestBean getTestBeanInstance() {
    return new TestBean();
  }
}
```

Tak zdefiniowana statyczna metoda fabrykująca może zwracać egzemplarze obiektów dowolnego typu; klasa zwracanego egzemplarza nie musi być tożsama z klasą zawierającą samą metodę fabrykującą.

Trzecim mechanizmem tworzenia nowych egzemplarzy komponentu jest wywoływanie niestatycznych metod fabrykujących innych egzemplarzy komponentów w ramach tego samego kontenera:

```
...
<bean id="nonStaticFactory" class="ch02.sample4.NonStaticFactory"/>

<bean id="testBeanObtainedViaNonStaticFactory"
      factory-bean="nonStaticFactory" factory-method="getTestBeanInstance"/>
...
```

```
public class NonStaticFactory {
  public TestBean getTestBeanInstance() {
    return new TestBean();
  }
}
```

Kiedy będzie potrzebny nowy egzemplarz komponentu `testBeanObtainedViaNonStaticFactory`, kontener w pierwszej kolejności utworzy egzemplarz fabryki `nonStaticFactory` i wywoła udostępnianą przez tę fabrykę metodę `getTestBeanInstance()`. Zauważ, że w tym przypadku w ogóle nie określiliśmy wartości atrybutu `class`.

Kiedy już uda się uzyskać nowy egzemplarz obiektu, kontener traktuje go dokładnie tak samo jak wszystkie pozostałe egzemplarze, niezależnie od tego, czy został utworzony za pośrednictwem konstruktora, statycznej metody fabrykującej czy metody fabrykującej innego egzemplarza. Oznacza to, że każdy z tych egzemplarzy może być przedmiotem wstrzykiwania zależności przez metody ustawiające i podlega normalnemu cyklowi życia (w tym odpowiednim wywołaniom zwrotnym).

Komponenty singletonowe kontra komponenty niesingletonowe (prototypowe)

Ważnym aspektem cyklu życia komponentu jest to, czy kontener traktuje go jak singleton czy jak zwykłą klasę z wieloma egzemplarzami. Domyślne komponenty singletonowe są tworzone przez kontener tylko raz. Kontener następnie przechowuje i wykorzystuje ten sam egzemplarz komponentu za każdym razem, gdy następuje odwołanie do danego komponentu. Takie rozwiązanie może oznaczać znaczne oszczędności zasobów (w szczególności pamięci, ale także obciążenia procesora) w porównaniu z tworzeniem nowego egzemplarza komponentu w odpowiedzi na każde kolejne żądanie. Komponenty singletonowe są więc najlepszym rozwiązaniem zawsze wtedy, gdy tylko implementacja właściwych klas dopuszcza taką możliwość; czyli wtedy, gdy komponent jest bezstanowy lub gdy jego stan jest ustawiany tylko raz, w czasie inicjalizacji, i — tym samym — **zapewnia bezpieczeństwo wątków** (może być wykorzystywany jednocześnie przez więcej niż jeden wątek). Komponenty singletonowe są rozwiązaniem domyślnym, ponieważ zdecydowana większość praktycznych usług, kontrolerów i zasobów konfigurowanych w ramach kontenera i tak jest implementowana w formie klas gwarantujących bezpieczną pracę w środowisku wielowątkowym, zatem nie modyfikują swojego stanu po wykonaniu fazy inicjalizacji.

Niesingletonowe, **prototypowe** komponenty, jak sama nazwa wskazuje, powinny być definiowane z atrybutem `singleton` równym `false`. Warto pamiętać, że cykl życia komponentu prototypowego często różni się od cyklu życia komponentu singletonowego. Kiedy kontener otrzymuje żądanie dostarczenia komponentu prototypowego, następuje oczywiście inicjalizacja i przekazanie odpowiedniego egzemplarza, ale na tym rola kontenera się kończy (od tego momentu kontener nie przechowuje zwróconego egzemplarza). O ile więc istnieje możliwość wymuszenia na kontenerze Springa wykonania pewnych operacji kończących cykl życia komponentów singletonowych (patrz dalsza część tego rozdziału), w przypadku komponentów prototypowych te same operacje będzie trzeba zrealizować w kodzie użytkownika, ponieważ kontener nie będzie już miał żadnej wiedzy o zwróconych egzemplarzach tych komponentów:

```
<bean id="singleton1" class="ch02.sample4.TestBean"/>

<bean id="singleton2" singleton="true" class="ch02.sample5.TestBean"/>

<bean id="prototype1" singleton="false" class="ch02.sample5.TestBean"/>
```

Definiowanie zależności komponentów

Spełnianie zależności komponentów (czy to w formie innych komponentów, czy tylko prostych wartości potrzebnych do działania danego komponentu) jest prawdziwym jądrem, kluczowym elementem funkcjonalności kontenera, warto więc dobrze zrozumieć rzeczywiste działanie tego procesu. Miałeś już okazję zapoznać się z przykładami podstawowych typów wstrzykiwania zależności, **wstrzykiwaniem przez konstruktory** i **wstrzykiwaniem przez metody ustawiające**, wiesz też, że Spring obsługuje obie formy tej techniki. Przekonałeś się również, jak Spring może wykorzystać odpowiednią metodę fabrykującą — zamiast używać konstruktora do uzyskiwania początkowego egzemplarza obiektu. Właśnie z uwagi na konieczność dostarczenia zależności do komponentu stosowanie metody fabrykującej do uzyskiwania egzemplarza komponentu można de facto uznać za odpowiednik tworzenia egzemplarza za pośrednictwem konstruktora. W przypadku użycia konstruktora to kontener odpowiada za zapewnienie wartości (opcjonalnych) argumentów konstruktora (które reprezentują zależności). Podobnie, w przypadku metody fabrykującej kontener dostarcza wartości (opcjonalnych) argumentów do metody fabrykującej (także reprezentujące zależności). Niezależnie od tego, czy początkowy egzemplarz obiektu jest tworzony przez konstruktor czy przez metodę fabrykującą, od tego momentu wszystkie egzemplarze są traktowane jednakowo.

Techniki wstrzykiwania przez konstruktory i wstrzykiwania przez metody ustawiające nie wykluczają się wzajemnie. Nawet jeśli Spring uzyskuje początkowy egzemplarz komponentu za pośrednictwem konstruktora lub metody fabrykującej i dostarcza wartości argumentów do konstruktora lub metody fabrykującej (wstrzykuje zależności), nadal może stosować mechanizm wstrzykiwania przez metody ustawiające do wprowadzania kolejnych zależności. Takie rozwiązanie może być szczególnie przydatne np. wtedy, gdy będziemy musieli użyć i zainicjalizować istniejącą klasę, która z jednej strony zawiera konstruktor pobierający jeden lub wiele argumentów i generujący komponent w znanym (prawidłowym) stanie początkowym, ale z drugiej strony bazuje na metodach ustawiających JavaBeans dla części opcjonalnych właściwości. Gdyby nie obsługa obu form wstrzykiwania zależności, nie byłbyś w stanie prawidłowo inicjalizować tego rodzaju obiektów w sytuacji, gdyby któraś z nich wymagała ustawienia opcjonalnych właściwości.

Przeanalizujmy teraz sposób, w jaki kontener inicjalizuje i realizuje zależności komponentów:

- Kontener w pierwszej kolejności inicjalizuje definicję komponentu bez tworzenia jego egzemplarza — zwykle następuje to w czasie uruchamiania samego kontenera. Zależności komponentu mogą być wyrażane wprost, w formie argumentów konstruktora lub metody fabrykującej i (lub) właściwości komponentu.
- Każda właściwość lub argument konstruktora w definicji komponentu ma albo postać wartości wymagającej ustawienia, albo odwołania do innego komponentu w ramach danej fabryki komponentów lub nadrzędnej fabryki komponentów.

- Kontener wykona tyle operacji sprawdzających poprawność definiowanych zależności, ile będzie mógł w czasie inicjalizacji definicji komponentu. Jeśli korzystasz z formatu konfiguracji XML, w przypadku niezgodności konfiguracji z typem XML DTD w pierwszej kolejności otrzymamy wyjątek wygenerowany przez parser XML. Nawet jeśli konfiguracja w formacie XML jest poprawna z punktu widzenia definicji typów DTD, w razie wykrycia logicznej niespójności przez Springa także otrzymamy odpowiedni wyjątek — przykładowo, dwie właściwości mogą się wzajemnie wykluczać, czego nie da się wykazać wyłącznie na podstawie analizy typów DTD.
- Jeśli zależność komponentu nie może być zrealizowana w praktyce (jeśli zależność komponentu ma postać innego komponentu, który nie istnieje) lub jeśli argument konstruktora lub wartość właściwości nie może zostać prawidłowo ustawiona, otrzymamy komunikat o błędzie tylko wtedy, gdy dany kontener rzeczywiście będzie musiał uzyskać nowy egzemplarz tego komponentu i wstrzyknąć jego zależności. Jeśli okaże się, że egzemplarz komponentu nigdy nie będzie potrzebny, nie można wykluczyć, że ewentualne błędy w definicji zależności komponentu lub samego komponentu nigdy nie zostaną wykryte (przynajmniej do momentu jego użycia). Między innymi po to, aby umożliwić możliwie szybkie wykrywanie błędów, konteksty aplikacji (ale nie fabryki komponentów) domyślnie tworzą wstępne egzemplarze komponentów singletonowych. Faza wstępnego tworzenia egzemplarzy obejmuje iteracyjne przeszukiwanie wszystkich komponentów singletonowych (w ich domyślnych stanach), utworzenie egzemplarzy każdej ze wstrzykiwanych zależności i umieszczenie tych egzemplarzy w pamięci podręcznej. Warto pamiętać, że etap wstępnego tworzenia egzemplarzy komponentów można zmodyfikować albo za pomocą atrybutu `default-lazy-init` elementu `beans` definiowanego na najwyższym poziomie pliku konfiguracji, albo na poziomie poszczególnych komponentów (elementów `bean`) za pośrednictwem atrybutu `lazy-init`.
- Kiedy kontener potrzebuje nowego egzemplarza danego komponentu (zazwyczaj w wyniku wywołania metody `getBean()` lub odwołania ze strony innego komponentu, dla którego bieżący komponent jest zależnością), uzyskuje początkowy egzemplarz za pośrednictwem konfigurowanego konstruktora lub metody fabrykującej, po czym podejmuje próby wstrzykiwania zależności, opcjonalnych argumentów konstruktora lub metody fabrykującej oraz opcjonalnych wartości właściwości.
- Argumenty konstruktora lub właściwości komponentu, które odwołują się do innego komponentu, w pierwszej kolejności wymuszają na kontenerze tworzenie lub uzyskiwanie dostępu do tamtego komponentu. W efekcie wskazywany komponent jest w istocie zależnością komponentu, który się do niego odwołuje. Operacje tworzenia lub uzyskiwania dostępu do komponentów składają się na logicznie spójny łańcuch, który można reprezentować w formie grafu zależności.
- Każdy argument konstruktora lub wartość właściwości musi zapewniać możliwość konwersji z oryginalnego typu lub formatu na typ faktycznie oczekiwany przez argument konstruktora lub właściwość komponentu (oczywiście jeśli oba typy są różne). Spring oferuje mechanizmy konwersji argumentów przekazywanych w formacie łańcuchowym do wszystkich wbudowanych typów skalarnych, a więc `int`, `long`, `boolean` itd., oraz wszystkich typów opakowań, w tym `Integer`, `Long`, `Boolean` itd. Spring wykorzystuje też implementacje interfejsu `PropertyEditor` komponentów JavaBeans do konwersji wartości typu `String` na niemal dowolne typy danych. Kontener automatycznie rejestruje i wykorzystuje wiele różnych implementacji

tego interfejsu. Przykładem takiej implementacji jest klasa `ClassEditor`, która konwertuje łańcuch reprezentujący nazwę klasy na egzemplarz klasy `Class`, który może być przekazany do właściwości oczekującej takiego egzemplarza; innym przykładem jest klasa `ResourceEditor`, która konwertuje łańcuch reprezentujący ścieżkę lokalizacji na obiekt klasy `Resource` Springa, który może być wykorzystywany do abstrakcyjnego uzyskiwania dostępu do zasobów. Wszystkie wbudowane edytory właściwości omówimy w następnym rozdziale. Istnieje też możliwość rejestrowania własnych implementacji interfejsu `PropertyEditor` obsługujących Twoje typy niestandardowe (patrz podpunkt „Tworzenie własnych edytorów właściwości" w dalszej części tego rozdziału).

- Także wariant konfiguracji oparty na formacie XML, który jest wykorzystywany przez większość implementacji fabryk komponentów i kontekstów aplikacji, wykorzystuje własne elementy i atrybuty umożliwiające definiowanie złożonych kolekcji, czyli list, zbiorów, map i właściwości. Okazuje się, że wartości tych kolekcji mogą być dowolnie zagnieżdżane.
- Zależności mogą też mieć charakter niejawny — w ekstremalnych przypadkach Spring może użyć refleksji do sprawdzenia argumentów pobieranych przez konstruktor komponentu (lub wartości właściwości tego komponentu) nawet wtedy, gdy nie zadeklarowano odpowiednich zależności. Kontener może na podstawie takiej procedury zbudować listę prawidłowych zależności danego komponentu. Następnie, korzystając z mechanizmu nazywanego **wiązaniem automatycznym** (ang. *autowiring*), kontener może te zależności wypełnić w oparciu o dopasowania według **typów** lub **nazw**. Na razie pominiemy temat automatycznego wiązania, ale z pewnością do niego wrócimy w dalszej części tego rozdziału.

Definiowanie zależności komponentów w szczegółach

W tym podpunkcie szczegółowo omówimy techniki definiowania wartości właściwości komponentów i argumentów konstruktorów w formacie XML. Każdy element `bean` może zawierać zero, jeden lub wiele elementów `constructor-arg` określających argumenty konstruktora lub metody wyszukującej. Elementy `bean` mogą też zawierać zero, jeden lub wiele elementów `property` określających właściwości JavaBean wymagających ustawienia. Jeśli sytuacja tego wymaga, można łączyć oba podejścia (argumenty konstruktora i właściwości JavaBean) — takie rozwiązanie zastosowano w poniższym przykładzie, gdzie argument konstruktora jest w istocie odwołaniem do innego komponentu, natomiast właściwość typu `int` jest zwykłą wartością:

```
<beans>
  <bean id="weatherService" class="ch02.sample6.WeatherServiceImpl">
<constructor-arg index="0">
      <ref local="weatherDao"/>
    </constructor-arg>
    <property name="maxRetryAttempts"><value>2</value></property>
</bean>

  <bean id="weatherDao" class="ch02.sample6.StaticDataWeatherDaoImpl">
  </bean>
</beans>
```

Analiza wspominanego już dokumentu XML DTD pokazuje, że w ramach elementów property i constructor-arg można stosować wiele innych elementów. Każdy z tych elementów definiuje jakiś typ wartości deklarowanej właściwości lub argumentu konstruktora:

```
(bean | ref | idref | list | set | map | props | value | null)
```

Element ref służy do takiego ustawiania wartości właściwości lub argumentu konstruktora, aby odwoływała się do innego komponentu w ramach tej samej fabryki komponentów lub w ramach jej fabryki nadrzędnej:

```
<ref local="weatherDao"/>
<ref bean="weatherDao"/>
<ref parent="weatherDao"/>
```

Atrybuty local, bean i parent wzajemnie się wykluczają i muszą zawierać identyfikator innego komponentu. W przypadku użycia atrybutu local analizator składni dokumentu XML może zweryfikować istnienie wskazanego komponentu już na etapie analizy składniowej. Ponieważ jednak całość bazuje na mechanizmie IDREF języka XML, komponent musi być zdefiniowany w tym samym pliku XML co odwołanie do tego komponentu, a jego definicja musi wykorzystywać atrybut id określający identyfikator, do którego będziemy się odwoływali (zamiast atrybutu name). W przypadku użycia atrybutu bean wskazany komponent może się znajdować w tym samym lub innym fragmencie dokumentu XML wykorzystywanym albo do budowy definicji fabryki komponentów, albo w definicji fabryki nadrzędnej względem fabryki bieżącej. Za weryfikację istnienia konkretnych komponentów może co prawda odpowiadać sam Spring (nie analizator składniowy XML), ale tylko wówczas, gdy dana zależność rzeczywiście musi zostać zrealizowana (a więc nie w czasie ładowania fabryki komponentów). Znacznie rzadziej stosowany atrybut parent określa, że komponent docelowy musi pochodzić z fabryki nadrzędnej względem bieżącej fabryki komponentów. Takie rozwiązanie może być przydatne w nieczęstych sytuacjach, w których występuje konflikt nazw komponentów w bieżącej i nadrzędnej fabryce komponentów.

Element value służy do określania wartości prostych właściwości lub argumentów konstruktora. Jak już wspominano, niezbędnym krokiem jest konwersja wartości źródłowej (która ma postać łańcucha) na odpowiedni typ docelowej właściwości lub argumentu konstruktora, czyli dowolny wbudowany typ skalarny, odpowiedni typ opakowania lub dowolny inny typ, dla którego w kontenerze zarejestrowano implementację interfejsu PropertyEditor zdolną do obsługi tego typu. Przeanalizujmy teraz konkretny przykład:

```
<property name="classname">
  <value>ch02.sample6.StaticDataWeatherDaoImpl</value>
</property>
```

Powyższy fragment kodu ustawia właściwość typu String nazwaną classname i przypisuje mu stałą wartość ch02.sample6.StaticDataWeatherDaoImpl; gdyby jednak właściwość classname była typu java.lang.Class, fabryka komponentów musiałaby użyć wbudowanej (i automatycznie rejestrowanej) implementacji interfejsu PropertyEditor (w tym przypadku klasy ClassEditor) do konwersji tej wartości łańcuchowej na egzemplarz obiektu klasy Class.

Istnieje możliwość stosunkowo prostego rejestrowania własnych, niestandardowych implementacji interfejsu PropertyEditor, które będą obsługiwały konwersję łańcuchów na dowolne inne typy danych niezbędne do właściwej konfiguracji kontenera. Dobrym przykładem

sytuacji, w której takie rozwiązanie jest uzasadnione, jest przekazywanie łańcuchów reprezentujących daty, które mają być następnie wykorzystywane do ustawiania właściwości typu `Date`. Ponieważ daty są szczególnie wrażliwe na uwarunkowania regionalne, użycie odpowiedniej implementacji interfejsu `PropertyEditor`, która będzie prawidłowo obsługiwała łańcuch źródłowy, jest najprostszym rozwiązaniem tego problemu. Sposób rejestrowania niestandardowych implementacji tego interfejsu zostanie przedstawiony w dalszej części tego rozdziału, przy okazji omawiania klasy `CustomEditorConfigurer` i **postprocesora** fabryki komponentów. Niewielu programistów zdaje sobie sprawę z tego, że odpowiednie mechanizmy interfejsu `PropertyEditor` automatycznie wykrywają i wykorzystują wszystkie implementacje tego interfejsu, które należą do tego samego pakietu co klasa przeznaczona do konwersji (jedynym warunkiem jest zgodność nazwy tej klasy z nazwą klasy implementującej wspomniany interfejs, która dodatkowo musi zawierać sufiks `Editor`). Oznacza to, że w przypadku klasy `MyType` implementacja interfejsu `PropertyEditor` nazwana `MyTypeEditor` i należąca do tego samego pakietu co klasa `MyType` zostanie automatycznie wykryta i użyta przez kod pomocniczy JavaBeans zdefiniowany w odpowiedniej bibliotece Javy (bez najmniejszego udziału Springa).

Właściwości lub argumenty konstruktora, którym należy przypisać wartość `null`, wymagają specjalnego traktowania, ponieważ pusty element `value` jest interpretowany jak łańcuch pusty. Zamiast braku wartości należy więc użyć elementu `null`:

```
<property name="optionalDescription"><null/></property>
```

Element `idref` jest wygodnym sposobem wychwytywania błędów w odwołaniach do innych komponentów za pośrednictwem wartości łańcuchowych reprezentujących ich nazwy. Istnieje kilka komponentów pomocniczych samego Springa, które odwołują się do innych komponentów (i wykonują za ich pomocą pewne działania) właśnie w formie wartości swoich właściwości. Wartości tego rodzaju właściwości zwykle definiuje się w następujący sposób:

```
<property name="beanName"><value>weatherService</value></property>
```

Możliwie szybkie wykrywanie ewentualnych literówek w podobnych odwołaniach byłoby oczywiście korzystne — element `idref` w praktyce odpowiada właśnie za taką weryfikację. Użycie elementu `property` w postaci:

```
<property name="beanName"><idref local="weatherService"/></property>
```

umożliwia analizatorowi składniowemu XML udział we wczesnym procesie weryfikacji, ponieważ już na etapie analizy składniowej można bez trudu wykryć odwołania do komponentów, które w rzeczywistości nie istnieją. Wartość wynikowa tej właściwości będzie dokładnie taka sama jak w przypadku użycia standardowego znacznika `value`.

Elementy `list`, `set`, `map` i `props` umożliwiają definiowanie i ustawianie złożonych właściwości lub argumentów konstruktorów (odpowiednio typów `java.util.List`, `java.util.Set`, `java.util.Map` i `java.util.Properties`). Przeanalizujmy teraz całkowicie nierzeczywisty przykład, w którym definiowany komponent JavaBean będzie zawierał po jednej właściwości każdego z wymienionych typów złożonych:

```
<beans>
  <bean id="collectionsExample" class="ch02.sample7.CollectionsBean">
    <property name="theList">
```

```
        <list>
          <value>red</value>
          <value>red</value>
          <value>blue</value>
          <ref local="curDate"/>
          <list>
            <value>one</value>
            <value>two</value>
            <value>three</value>
          </list>
        </list>
      </property>
      <property name="theSet">
  <set>
          <value>red</value>
          <value>red</value>
          <value>blue</value>
        </set>
  </property>
      <property name="theMap">
  <map>
          <entry key="left">
            <value>right</value>
          </entry>
          <entry key="up">
            <value>down</value>
          </entry>
          <entry key="date">
            <ref local="curDate"/>
          </entry>
        </map>
  </property>
      <property name="theProperties">
  <props>
          <prop key="left">right</prop>
          <prop key="up">down</prop>
        </props>
  </property>
    </bean>

    <bean id="curDate" class="java.util.GregorianCalendar"/>
  </beans>
```

Kolekcje typu `List`, `Map` i `Set` mogą zawierać dowolne spośród wymienionych poniżej elementów:

```
  (bean | ref | idref | list | set | map | props | value | null)
```

Jak pokazuje przedstawiony przykład listy, typy kolekcji mogą być dowolnie zagnieżdżane. Warto jednak pamiętać, że właściwości lub argumenty konstruktorów otrzymujące na wejściu typy kolekcji muszą być deklarowane za pomocą typów `java.util.List`, `java.util.Set` lub `java.util.Map`. Nie możesz stosować innych typów kolekcji (np. `ArrayList`), nawet jeśli są obsługiwane w Springu. Tego rodzaju ograniczenia mogą stanowić poważny problem, jeśli musimy zapewnić wartość dla właściwości istniejącej klasy, która oczekuje określonego

typu — w takim przypadku nie można konkretnego typu kolekcji zastąpić jego uniwersalnym odpowiednikiem. Jednym z rozwiązań jest użycie udostępnianych przez Springa pomocniczych **komponentów fabrykujących** (ang. *factory beans*): `ListFactoryBean`, `SetFactoryBean` lub `MapFactoryBean`. Za ich pomocą można swobodnie określać docelowy typ kolekcji, w tym np. `java.util.LinkedList`. Więcej informacji na ten temat znajdziesz w dokumentacji JavaDoc Springa. Same komponenty fabrykujące omówimy w dalszej części rozdziału.

Ostatnim elementem, który może występować w roli wartości właściwości lub wartości argumentu konstruktora (lub wewnątrz któregoś z opisanych przed chwilą elementów kolekcji), jest element `bean`. Oznacza to, że definicja komponentów może być de facto zagnieżdżana w definicji innego komponentu (jako właściwość komponentu zewnętrznego). Przeanalizujmy teraz przebudowany przykład wstrzykiwania przez metodę ustawiającą:

```
<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <bean class="ch02.sample2.StaticDataWeatherDaoImpl"/>
        ...
      </bean>
    </property>
  </bean>
</beans>
```

Zagnieżdżone definicje komponentów są szczególnie przydatne w sytuacji, gdy komponent wewnętrzny nie znajduje żadnych zastosowań poza zakresem komponentu zewnętrznego. W porównaniu z poprzednią wersją tego przykładu obiekt DAO (który ustawiono jako zależność usługi pogodowej) został włączony do komponentu usługi pogodowej i przyjął postać jego komponentu wewnętrznego. Żaden inny komponent ani użytkownik zewnętrzny nie będzie przecież potrzebował tego obiektu DAO, zatem utrzymywanie go poza zakresem komponentu usługi (jako osobnej, zewnętrznej definicji) mijałoby się z celem. Użycie komponentu wewnętrznego jest w tym przypadku rozwiązaniem bardziej zwięzłym i klarownym. Komponent wewnętrzny nie musi mieć zdefiniowanego identyfikatora, choć nie jest to zabronione. *Uwaga:* Komponenty wewnętrzne zawsze są komponentami prototypowymi, a ewentualny atrybut `singleton` jest ignorowany. W tym przypadku nie ma to najmniejszego znaczenia — ponieważ istnieje tylko jeden egzemplarz komponentu zewnętrznego (który jest singletonem), utworzenie więcej niż jednego egzemplarza komponentu wewnętrznego jest wykluczone (niezależnie od tego, czy zadeklarujemy ten komponent jako singletonowy czy jako prototypowy). Gdyby jednak prototypowy komponent zewnętrzny potrzebował zależności singletonowej, nie powinniśmy tej zależności definiować w formie komponentu wewnętrznego, tylko jako odwołanie do singletonowego komponentu zewnętrznego.

Samodzielne (ręczne) deklarowanie zależności

Kiedy właściwość komponentu lub argument konstruktora odwołuje się do innego komponentu, mamy do czynienia z deklaracją zależności od tego zewnętrznego komponentu. W niektórych sytuacjach konieczne jest wymuszenie inicjalizacji jednego komponentu przed innym, nawet jeśli ten drugi nie został zadeklarowany jako właściwość pierwszego. Wymuszanie określonej kolejności inicjalizacji jest uzasadnione także w innych przypadkach, np. kiedy

dana klasa wykonuje jakieś statyczne operacje inicjalizujące w czasie wczytywania. Przykładowo, sterowniki baz danych zwykle są rejestrowane w egzemplarzu interfejsu `DriverManager` biblioteki JDBC. Do ręcznego określania zależności pomiędzy komponentami służy atrybut `depends-on`, który wymusza utworzenie egzemplarza innego komponentu jeszcze przed uzyskaniem dostępu do komponentu zależnego. Poniższy przykład pokazuje, jak można wymuszać wczytywanie sterownika bazy danych:

```
<bean id="load-jdbc-driver" class="oracle.jdbc.driver.OracleDriver"/>

<bean id="weatherService" depends-on="load-jdbc-driver" class="...">
  ...
</bean>
```

Warto pamiętać, że większość pul połączeń z bazą danych oraz klas pomocniczych Springa (np. `DriverManagerDataSource`) korzysta z tego mechanizmu wymuszania wczytywania, zatem powyższy fragment kodu jest tylko przykładem popularnego rozwiązania.

Automatyczne wiązanie zależności

Do tej pory mieliśmy do czynienia z deklaracjami zależności komponentów wyrażanymi wprost (za pośrednictwem wartości właściwości i argumentów konstruktora). W pewnych okolicznościach Spring może użyć mechanizmu introspekcji klas komponentów w ramach bieżącej fabryki i przeprowadzić **automatyczne wiązanie** zależności. W takim przypadku właściwość komponentu lub argument kontenera nie muszą być deklarowane (np. w pliku XML), ponieważ Spring użyje refleksji do odnalezienia typu i nazwy odpowiedniej właściwości, po czym dopasuje ją do innego komponentu w danej fabryce (według jego typu lub nazwy). Takie rozwiązanie może co prawda oszczędzić programiście mnóstwo pracy związanej z samodzielnym przygotowywaniem odpowiedniego kodu, jednak niewątpliwym kosztem tego podejścia jest mniejsza przejrzystość. Mechanizm automatycznego wiązania zależności można kontrolować zarówno na poziomie całego kontenera, jak i na poziomie definicji poszczególnych komponentów. Ponieważ nieostrożne korzystanie z tej techniki może prowadzić do nieprzewidywalnych rezultatów, mechanizm automatycznego wiązania jest domyślnie wyłączony. Automatyczne wiązanie na poziomie komponentów jest kontrolowane za pośrednictwem atrybutu `autowire`, który może zawierać pięć wartości:

- `no` — mechanizm automatycznego wiązania zależności w ogóle nie będzie stosowany dla danego komponentu. Właściwości tego komponentu oraz argumenty konstruktora muszą być deklarowane wprost, a wszelkie odwołania do innych komponentów wymagają używania elementu `ref`. Okazuje się, że jest to domyślny sposób obsługi poszczególnych komponentów (przynajmniej jeśli domyślne ustawienia na poziomie fabryki komponentów nie zostały zmienione). Opisany tryb jest zalecany w większości sytuacji, szczególnie w przypadku większych wdrożeń, gdzie zależności deklarowane wprost stanowią swoistą dokumentację struktury oprogramowania i są dużo bardziej przejrzyste.
- `byName` — wymusza automatyczne wiązanie zależności według nazw właściwości. Nazwy właściwości są wykorzystywane podczas odnajdywania dopasowań do komponentów w bieżącej fabryce komponentów. Przykładowo, jeśli dana właściwość ma przypisaną nazwę `weatherDao`, wówczas kontener spróbuje ustawić tę właściwość jako odwołanie do innego komponentu nazwanego właśnie `weatherDao`.

Jeśli taki pasujący komponent nie zostanie znaleziony, właściwość pozostanie nieustawiona. Taka reakcja na brak dopasowania pomiędzy nazwą właściwości a nazwą komponentu jest opcjonalna — jeśli chcesz traktować brak dopasowania jak błąd, możesz do definicji komponentu dodać atrybut `dependency-check="objects"` (patrz dalsza część tego rozdziału).

- `byType` — automatyczne wiązanie zależności przez dopasowywanie typów. Podobne rozwiązanie zastosowano w kontenerze PicoContainer, innym popularnym produkcie obsługującym wstrzykiwanie zależności. W przypadku każdej właściwości — jeśli w bieżącej fabryce komponentów istnieje dokładnie jeden komponent tego samego typu co ta właściwość — wartość jest ustawiana właśnie jako ten komponent. Jeśli w fabryce istnieje więcej niż jeden komponent pasującego typu, efektem nieudanej próby dopasowania jest błąd krytyczny, który zwykle prowadzi do wygenerowania odpowiedniego wyjątku. Tak jak w przypadku automatycznego wiązania zależności według nazwy (wartość `byName`) w przypadku braku pasujących komponentów właściwość pozostaje nieustawiona. Jeśli brak dopasowania powinien być traktowany jak błąd, wystarczy do definicji komponentu dodać atrybut `dependency-check="objects"` (patrz dalsza część tego rozdziału).
- `constructor` — automatyczne wiązanie zależności według typów argumentów konstruktora. Ten tryb bardzo przypomina wiązanie zależności z właściwościami, tyle że polega na szukaniu w fabryce dokładnie jednego pasującego (według typu) komponentu dla każdego z argumentów konstruktora. W przypadku wielu konstruktorów Spring w sposób zachłanny będzie próbował spełnić wymagania konstruktora z największą liczbą pasujących argumentów.
- `autodetect` — wybiera tryby `byType` i `constructor` w zależności od tego, który z nich lepiej pasuje do danej sytuacji. W przypadku wykrycia domyślnego, bezargumentowego konstruktora stosowany jest tryb `byType`; w przeciwnym razie zostanie zastosowany tryb `constructor`.

Istnieje możliwość ustawienia innego domyślnego trybu automatycznego wiązania zależności (innego niż `no`) dla wszystkich komponentów w danej fabryce — wystarczy użyć atrybutu `default-autowire` w elemencie `beans` na najwyższym poziomie dokumentu XML. Warto też pamiętać o możliwości łączenia technik automatycznego wiązania z wiązaniem wprost, wówczas elementy definiujące zależności (`property` lub `contructor-arg`) mają wyższy priorytet od zależności wykrywanych automatycznie.

Sprawdźmy teraz, jak można użyć mechanizmu automatycznego wiązania zależności dla naszej usługi pogodowej i jej obiektu DAO. Jak widać, możemy wyeliminować z definicji komponentu właściwość `weatherDao` i włączyć automatyczne wiązanie według nazw, a mimo to Spring nadal będzie w stanie odnaleźć wartość właściwości wyłącznie w oparciu o dopasowanie jego nazwy. W tym przypadku moglibyśmy użyć także trybu automatycznego wiązania zależności według typu, ponieważ do typu naszej właściwość (`WeatherDao`) pasuje tylko jeden komponent w kontenerze:

```
<beans>
  <bean id="weatherService" autowire="byName"
        class="ch02.sample2.WeatherServiceImpl">
    <!-- brak deklaracji właściwości weatherDao -->
  </bean>
```

```
    <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
    </bean>
</beans>
```

Korzystanie z technik automatycznego wiązania zależności jest o tyle kuszące, że zwalnia programistę z obowiązku pisania znacznej ilości kodu konfiguracji fabryki komponentów, jednak warto zachować należytą ostrożność i stosować ten mechanizm z rozwagą.

> Usuwając zależności deklarowane wprost, rezygnujemy z jednej z form dokumentowania tych zależności. Co więcej, ze stosowaniem trybów `byType` lub nawet `byName` wiąże się ryzyko nieprzewidywalnych zachowań w sytuacji, gdy uda się odnaleźć więcej niż jedno dopasowanie (pasujący komponent) lub w razie braku jakiegokolwiek dopasowania. Szczególnie w przypadku większych, bardziej skomplikowanych wdrożeń zaleca się unikanie automatycznego wiązania zależności szerokim łukiem lub przynajmniej bardzo rozważne korzystanie z tego mechanizmu, ponieważ w przeciwnym razie może się okazać, że ograniczając ilość kodu XML, dodatkowo skomplikowaliśmy całą aplikację. Większość współczesnych środowisk IDE oferuje (wbudowane lub dostępne w formie modułów rozszerzeń) edytory XML z obsługą DTD, które mogą oszczędzić mnóstwo czasu i wysiłku programisty podczas tworzenia konfiguracji komponentów, zatem rozmiar deklaracji zależności wyrażonych wprost nie stanowi wielkiego problemu. Tym, co może się doskonale sprawdzać w pewnych sytuacjach, jest stosowanie automatycznego wiązania zależności dla prostych, niskopoziomowych „wnętrzności" (np. `DataSource`) i jednocześnie deklarowanie zależności wprost w przypadku bardziej skomplikowanych aspektów. W ten sposób można wyeliminować nadmiar pracy bez konieczności poświęcania transparentności.

Dopasowywanie argumentów konstruktora

Ogólna zasada mówi, że dla każdego deklarowanego elementu `constructor-arg` należy dodatkowo stosować opcjonalne atrybuty `index` i (lub) `type`. Chociaż oba wymienione atrybuty są opcjonalne, w przypadku pominięcia choćby jednego z nich zadeklarowana lista argumentów konstruktora będzie dopasowywana do rzeczywistych argumentów według ich typów. Jeśli zadeklarowane argumenty są odwołaniami do innych typów (np. skomplikowanych komponentów lub takich typów złożonych jak `java.lang.Map`), kontener będzie mógł znaleźć odpowiednie dopasowanie stosunkowo łatwo, szczególnie w sytuacji, gdy będzie istniał tylko jeden konstruktor. Jeśli jednak zadeklarujemy wiele argumentów tego samego typu lub użyjemy znacznika `value`, który będzie de facto wartością bez typu (zadeklarowaną w formie łańcucha), próby automatycznego dopasowywania mogą się zakończyć błędami lub innymi nieprzewidywalnymi zachowaniami.

Przeanalizujmy przykład komponentu, który zawiera pojedynczy konstruktor pobierający numeryczną wartość kodu błędu oraz łańcuchową wartość komunikatu o błędzie. Jeśli spróbujemy użyć znacznika `<value>` do dostarczania wartości dla tych argumentów, będziemy musieli przekazać kontenerowi jakąś dodatkową wskazówkę odnośnie realizacji tego zadania. Możemy posłużyć się albo atrybutem `index` do określenia właściwego indeksu argumentu (liczonego od zera), co ułatwi dopasowanie przekazywanej wartości do odpowiedniego argumentu konstruktora:

```
<beans>
  <bean id="errorBean" class="ch02.sampleX.ErrorBean">
    <constructor-arg index="0"><value>1000</value></constructor-arg>
```

```
    <constructor-arg index="1"><value>Nieoczekiwany błąd</value></constructor-arg>
  </bean>
</beans>
```

albo dostarczyć kontenerowi taką ilość informacji, która pozwoli mu właściwie przeprowadzić proces dopasowania w oparciu o typy danych — wówczas powinniśmy użyć atrybutu `type` określającego typ danej wartości:

```
<beans>
  <bean id="errorBean" class="ch02.sampleX.ErrorBean">
    <constructor-arg type="int"><value>1000</value></constructor-arg>
    <constructor-arg type="java.lang.String">
      <value>Nieoczekiwany błąd</value>
    </constructor-arg>
  </bean>
</beans>
```

Sprawdzanie poprawności zależności komponentów

Często się zdarza, że część właściwości JavaBean danego obiektu ma charakter opcjonalny. Możesz, ale nie musisz, ustawiać ich wartości w zależności od tego, czy odpowiednia wartość jest potrzebna w konkretnym przypadku testowym; warto jednak pamiętać, że kontener nie dysponuje mechanizmami, które mogłyby Ci pomóc w wychwytywaniu błędów nieustawienia właściwości, które tego wymagają. Jeśli jednak masz do czynienia z komponentem, którego wszystkie właściwości (lub przynajmniej wszystkie właściwości określonego typu) muszą być ustawione, dużym ułatwieniem może być oferowany przez kontener mechanizm weryfikacji poprawności zależności. Jeśli zdecydujemy się na użycie tego mechanizmu, kontener będzie traktował ewentualny brak deklaracji wprost lub automatycznego wiązania zależności jako błąd. Kontener domyślnie nie podejmuje prób takiej weryfikacji (sprawdzania, czy wszystkie zależności są odpowiednio ustawione), można jednak zmienić to zachowanie w definicji komponentu za pomocą atrybutu `dependency-check`, który może mieć następujące wartości:

- `none` — brak weryfikacji zależności. Jeśli dla którejś z właściwości nie określono wartości, w przypadku braku weryfikacji taka sytuacja nie będzie traktowana jak błąd. Jest to domyślny sposób obsługi poszczególnych komponentów (przynajmniej jeśli ustawienia domyślne nie zostaną zmienione na poziomie fabryki komponentów).
- `simple` — sprawdza, czy ustawiono typy proste i kolekcje — brak wartości w tych właściwościach będzie traktowany jak błąd. Pozostałe właściwości mogą, ale nie muszą być ustawione.
- `objects` — sprawdza tylko właściwości innych typów niż typy proste i kolekcje — brak wartości w tych właściwościach będzie traktowany jak błąd. Pozostałe właściwości mogą, ale nie muszą być ustawione.
- `all` — sprawdza, czy ustawiono wartości wszystkich właściwości, w tym we właściwościach typów prostych, kolekcjach i właściwościach typów złożonych.

Istnieje możliwość zmiany domyślnego trybu sprawdzania zależności (ustawienie innego trybu niż `none`) dla wszystkich komponentów w ramach danej fabryki — wystarczy użyć atrybutu `default-dependency-check` w elemencie `beans` na najwyższym poziomie hierarchii.

Pamiętaj też, że także interfejs wywołań zwrotnych `InitializingBean` (patrz następny punkt) może być wykorzystywany do ręcznego sprawdzania poprawności zależności.

Zarządzanie cyklem życia komponentu

Komponent w fabryce może się charakteryzować zarówno bardzo prostym, jak i względnie skomplikowanym cyklem życia — wszystko zależy od tego, jaki jest zakres odpowiedzialności danego komponentu. Ponieważ mówimy o obiektach POJO, cykl życia komponentu nie musi obejmować działań wykraczających poza tworzenie i używanie danego obiektu. Istnieje jednak wiele sposobów zarządzania i obsługi bardziej złożonymi cyklami życia, z których większość koncentruje się wokół wywołań zwrotnych, gdzie sam komponent i zewnętrzne obiekty **obserwatorów** (nazywane **postprocesorami komponentu**) mogą obsługiwać wiele etapów procesów inicjalizacji i destrukcji. Przeanalizujmy teraz możliwe akcje kontenera (patrz poniższa tabela), które mogą mieć miejsce w cyklu życia komponentu zarządzanego przez kontener.

Akcja	Opis
Inicjalizacja rozpoczynająca się w momencie utworzenia egzemplarza komponentu	Egzemplarz nowego komponentu jest tworzony za pośrednictwem konstruktora lub metody fabrykującej (oba rozwiązania są równoważne). Proces tworzenia egzemplarza komponentu jest inicjowany przez wywołanie metody `getBean()` fabryki komponentów lub przez żądanie ze strony innego komponentu, którego egzemplarz już istnieje i który zawiera zależność od bieżącego komponentu.
Wstrzyknięcie zależności	Zależności są wstrzykiwane do nowo utworzonego egzemplarza komponentu (w ramach omówionej już procedury).
Wywołanie metody `setBeanName()`	Jeśli komponent implementuje opcjonalny interfejs `BeanNameAware`, wówczas metoda `setBeanName()` należąca właśnie do tego interfejsu jest wywoływana w celu nadania komponentowi jego głównego identyfikatora (zadeklarowanemu w definicji komponentu).
Wywołanie metody `setBeanFactory()`	Jeśli komponent implementuje opcjonalny interfejs `BeanFactoryAware`, referencję do fabryki, w ramach której ten komponent został wdrożony, można wprowadzić do komponentu, wywołując metodę `setBeanFactory()` tego interfejsu. Warto pamiętać, że ponieważ także konteksty aplikacji są fabrykami komponentów, wspomniana metoda będzie wywoływana również w przypadku komponentów wdrożonych w ramach kontekstu aplikacji (jednak wówczas komponent otrzyma odwołanie do fabryki wykorzystywanej wewnętrznie przez dany kontekst).
Wywołanie metody `setResourceLoader()`	Jeśli dany komponent implementuje opcjonalny interfejs `ResourceLoaderAware` i jest wdrażany w ramach kontekstu aplikacji, przedmiotem wywołania jest metoda `setResourceLoader()` tego interfejsu, z parametrem będącym kontekstem aplikacjim który implementuje interfejs `ResourceLoader` (to rozwiązanie omówimy już w następnym rozdziale).

Akcja	Opis
Wywołanie metody `setApplicationEvent-Publisher()`	Jeśli komponent implementuje opcjonalny interfejs `ApplicationEventPublisherAware` i jest wdrażany w ramach kontekstu aplikacji, przedmiotem wywołania jest metoda `setApplicationEventPublisher()` tego interfejsu, a parametrem jest kontekst aplikacji, który implementuje interfejs `ApplicationEventPublisher` (także to rozwiązanie omówimy w następnym rozdziale).
Wywołanie metody `setMessageSource()`	Jeśli komponent implementuje opcjonalny interfejs `MessageSourceAware` i jest wdrażany w ramach kontekstu aplikacji, przedmiotem wywołania jest należąca do tego interfejsu metoda `setResourceLoader()`, a przekazywany jest kontekst aplikacji, który implementuje interfejs `MessageSource` (także to rozwiązanie omówimy w następnym rozdziale).
Wywołanie metody `setApplicationContext()`	Jeśli dany komponent implementuje opcjonalny interfejs `ApplicationContextAware` i jest wdrażany w ramach kontekstu aplikacji, zapewnienie komponentowi referencji do tego kontekstu wymaga wywołania metody `setApplicationContext()`, która jest częścią interfejsu `ApplicationContextAware`.
Przekazanie postprocesorom komponentu wywołania zwrotnego „przed inicjalizacją”	Postprocesory komponentu (które omówimy w dalszej części tego rozdziału) są specjalnymi rozwiązaniami pomocniczymi rejestrowanymi przez aplikację wraz z fabrykami komponentów. Postprocesory otrzymują wraz z wywołaniami zwrotnymi (poprzedzającymi właściwą inicjalizację) komponent, który mogą dowolnie przetwarzać.
Wywołanie metody `afterPropertiesSet()`	Jeśli komponent implementuje interfejs `InitializingBean`, następuje wywołanie definiowanej przez ten interfejs metody `afterPropertiesSet()`, która umożliwia komponentowi samodzielną inicjalizację.
Wywołanie zadeklarowanej metody inicjalizującej	Jeśli definicja danego komponentu zawiera deklarację metody inicjalizującej (w formie odpowiedniej wartości atrybutu `init-method`), właśnie ta metoda zostanie wywołana, aby umożliwić komponentowi samodzielną inicjalizację.
Przekazanie postprocesorom komponentu wywołania zwrotnego „po inicjalizacji” z argumentem zawierającym egzemplarz komponentu	Każdy postprocesor komponentu otrzymuje wywołanie zwrotne „po inicjalizacji” z dołączonym (w formie argumentu) egzemplarzem komponentu, który — w razie potrzeby — może dowolnie przetwarzać.
Użycie komponentu	Na tym etapie egzemplarz komponentu jest przedmiotem właściwych operacji. Oznacza to, że komponent jest zwracany do kodu obiektu wywołującego metodę `getBean()` i wykorzystywany do ustawienia właściwości pozostałych komponentów (które wywołały bieżący komponent jako swoje zależności) itd. **Ważna uwaga:** Tylko cykl życia komponentów singletonowych jest od tego momentu śledzony przez kontener, natomiast właścicielami komponentów prototypowych są obiekty klienckie, które z nich korzystają. Kontener będzie więc uczestniczył w przyszłych zdarzeniach związanych tylko z cyklem życia komponentów singletonowych. Od tego momentu wszystkie komponenty prototypowe muszą być w całości zarządzane przez swoich klientów (dotyczy to także wywołania metody niszczącej).

Akcja	Opis
Rozpoczęcie procedury destrukcji komponentu	W ramach procesu kończenia pracy fabryki komponentów lub kontekstu aplikacji (który obejmuje trzy opisane poniżej kroki) należy zniszczyć wszystkie składowane w pamięci podręcznej komponenty singletonowe. Należy pamiętać, że komponenty są niszczone w kolejności odpowiadającej występującymi między nimi zależnościami (najczęściej w kolejności odwrotnej do inicjalizacji).
Przekazanie postprocesorom komponentu wywołania zwrotnego „zniszcz”	Każdy postprocesor komponentu implementujący interfejs `DestructionAwareBeanPostProcessor` otrzymuje wywołanie zwrotne, które umożliwia mu przygotowanie komponentu singletonowego do zniszczenia.
Wywołanie metody `destroy()`	Jeśli komponent singletonowy implementuje interfejs `DisposableBean`, wówczas jest wywoływana metoda `destroy()` tego interfejsu, która umożliwia komponentowi wykonanie wszystkich niezbędnych operacji zwalniania zasobów.
Wywołanie zadeklarowanej metody niszczącej	Jeśli definicja komponentu singletonowego zawiera deklarację metody niszczącej (w formie wartości atrybutu `destroy-method`), odpowiednia metoda jest wywoływana na samym końcu, a jej działania najczęściej sprowadzają się do zwalniania wszystkich zasobów, które tego wymagają.

Wywołania zwrotne inicjalizacji i destrukcji

Wspomniane przed chwilą metody inicjalizujące i niszczące mogą być wykorzystywane do wykonywania wszystkich niezbędnych operacji w zakresie rezerwowania i zwalniania zasobów dla danego komponentu. W przypadku istniejących klas, które zawierają już unikatowo nazwane metody inicjalizujące i niszczące, jedynym skutecznym rozwiązaniem jest użycie atrybutów `init-method` i `destroy-method` wymuszających na kontenerze wywoływanie tych metod we właściwych momentach — patrz poniższy przykład, gdzie konieczne jest wywołanie metody `close()` zastosowanej w implementacji `DataSource` (korzystającej z Jakarta Commons DBCP):

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>jdbc:oracle:thin:@db-server:1521:devdb</value>
  </property>
  <property name="username"><value>john</value></property>
  <property name="password"><value>password</value></property>
</bean>
```

Ogólnie nawet podczas budowania nowego oprogramowania zaleca się stosowanie atrybutów `init-method` i `destroy-method`, które zasygnalizują kontenerowi konieczność wywołania odpowiednich metod inicjalizujących i niszczących — takie rozwiązanie jest pod wieloma względami lepsze od implementowania przez sam komponent interfejsów `InitializingBean`

i `DisposableBean` Springa z ich metodami `afterPropertiesSet()` i `destroy()`. To drugie podejście jest z jednej strony wygodne, ponieważ wspomniane interfejsy są automatycznie wykrywane przez Springa, co oznacza, że także wymienione metody są wywoływane przez ten framework bez udziału programisty, z drugiej strony niepotrzebnie wiąże Twój kod z kontenerem Springa. Jeśli kod aplikacji jest związany ze Springiem także z innych powodów i w innych punktach, ta dodatkowa zależność nie stanowi większego problemu, a stosowanie interfejsów `InitializingBean` i `DisposableBean` jednocześnie jest niemałym ułatwieniem dla programisty. Okazuje się, że także kod samego Springa, który jest wdrażany w formie komponentów, często wykorzystuje oba interfejsy.

Jak już wspomniano, po zakończeniu fazy inicjalizacji niesingletonowe, prototypowe komponenty nie są składowane ani zarządzane w kontenerze. Oznacza to, że w przypadku tych komponentów Spring nie ma żadnych możliwości ani w zakresie wywoływania metod niszczących, ani w kwestii jakichkolwiek innych działań związanych z cyklem życia komponentów (właściwych tylko dla zarządzanych przez kontener komponentów singletonowych). Każda taka metoda musi być wywoływana z poziomu kodu użytkownika. Co więcej, postprocesory nie mogą przetwarzać komponentów znajdujących się w fazie niszczenia.

Wywołania zwrotne interfejsów BeanFactoryAware i ApplicationContextAware

Komponent, który z jakiegoś względu musi mieć dostęp do danych i mechanizmów swojej fabryki komponentów lub kontekstu aplikacji, może implementować odpowiednio interfejs `BeanFactoryAware` i `ApplicationContextAware`. Z przedstawionej przed chwilą tabeli prezentującej kolejność akcji podejmowanych przez kontener i związanych z cyklem życia komponentu wynika, że referencje do kontenerów są przekazywane do komponentów za pośrednictwem metod `setBeanFactory()` i `setApplicationContext()` (definiowanych przez wspomniane interfejsy). Ogólnie, większość kodu aplikacji w ogóle nie powinna do prawidłowego funkcjonowania potrzebować informacji o swoich kontenerach, jednak w kodzie ściśle związanym ze Springiem takie rozwiązanie może się okazać bardzo przydatne (tego rodzaju wywołania zwrotne są stosowane między innymi w klasach samego Springa). Jednym z przypadków, w których kod aplikacji może wymagać referencji do fabryki komponentu, jest komponent singletonowy współpracujący z komponentami prototypowymi. Ponieważ zależności komponentu singletonowego są do niego wstrzykiwane tylko raz, tak stosowany mechanizm wstrzykiwania wyklucza możliwość uzyskiwania nowych egzemplarzy komponentów prototypowych w czasie wykonywania właściwych zadań (a więc po fazie inicjalizacji). Oznacza to, że jednym ze skutecznych rozwiązań tego problemu jest bezpośredni dostęp do fabryki tego komponentu. Wydaje się jednak, że w większości przypadków jeszcze lepszym wyjściem jest użycie wspominanej już techniki wstrzykiwania metody wyszukującej, która dodatkowo zapewnia izolację klasy od mechanizmów Springa i nazwy komponentu prototypowego.

Abstrakcja dostępu do usług i zasobów

Istnieje co prawda dużo więcej zaawansowanych funkcji oferowanych przez fabryki komponentów i konteksty aplikacji, o których do tej pory nawet nie wspominaliśmy, jednak udało nam się przeanalizować niemal wszystkie niskopoziomowe elementy składowe niezbędne do skutecznego programowania i wdrażania rozwiązań IoC. Przekonaliśmy się, jak

obiekty aplikacji mogą realizować swoje zadania wyłącznie we współpracy z pozostałymi obiektami udostępnianymi przez kontener (za pośrednictwem odpowiednich interfejsów i abstrakcyjnych nadklas) bez konieczności dostosowywania swoich zachowań do implementacji lub kodu źródłowego tych obiektów. Wiedza przekazana do tej pory powinna Ci w zupełności wystarczyć do realizacji typowych scenariuszy korzystania z technik wstrzykiwania zależności.

Tym, co może jeszcze budzić pewne wątpliwości, jest rzeczywisty sposób uzyskiwania pozostałych obiektów współpracujących (o których wiesz, że mogą być konfigurowane i udostępniane na rozmaite sposoby). Aby zrozumieć, jak Spring chroni nas przed potencjalnymi zawiłościami i jak radzi sobie z transparentnym zarządzaniem tego rodzaju usługami, przeanalizujmy kilka ciekawych przykładów.

Wyobraź sobie nieco zmieniony obiekt DAO naszej usługi pogodowej, który zamiast wykorzystywać dane statyczne używa JDBC do uzyskiwania historycznych informacji pogodowych z bazy danych. Wstępna implementacja takiego rozwiązania może wykorzystywać oryginalny interfejs `DriverManager` JDBC 1.0 i za jego pośrednictwem uzyskiwać niezbędne połączenie — patrz metoda `find()`:

```
public WeatherData find(Date date) {
  // Zwróć uwagę, że w konstruktorze lub metodzie inicjalizującej użyliśmy
  // wywołania Class.forName(driverClassName) do zarejestrowania
  // sterownika JDBC. Wraz z nazwą użytkownika i hasła wspomniany
  // sterownik został skonfigurowany jako właściwości tego komponentu.
  try {
    Connection con = DriverManager.getConnection(url, username, password);
    // od tego miejsca możemy używać tego połączenia
    ...
```

Kiedy wdrożymy taki obiekt DAO w postaci komponentu wewnątrz kontenera Springa, będziemy mogli skorzystać z ułatwień (przede wszystkim mechanizmów wstrzykiwania przez metody ustawiające i wstrzykiwania przez konstruktor) w zakresie dostarczania wszystkich wartości wymaganych przez sterownik JDBC do nawiązania połączenia, czyli adresu URL, nazwy użytkownika i hasła. Nie mamy jednak zamiaru tworzyć puli połączeń (niezależnie od tego, czy będziemy korzystać ze środowiska J2EE czy z mniej zaawansowanego kontenera); nasze połączenie w założeniu nie ma podlegać żadnym procedurom zarządzania transakcjami (znanym z kontenera J2EE i stosowanym tylko w przypadku połączeń zarządzanych przez kontener).

Dość oczywistym rozwiązaniem w kwestii uzyskiwania niezbędnych połączeń jest użycie implementacji interfejsu `DataSource` wprowadzonych w JDBC 2.0. Kiedy nasz obiekt DAO będzie już zawierał egzemplarz takiej implementacji, wystarczy że zażąda odpowiedniego połączenia (faktyczna realizacja tego żądania następuje gdzie indziej). Teoretyczna dostępność egzemplarza `DataSource` nie stanowi problemu, ponieważ wiemy, że istnieją takie niezależne, autonomiczne implementacje puli połączeń jak Apache Jakarta Commons DBCP, które można z powodzeniem stosować w środowiskach J2SE oraz J2EE i które są udostępniane za pośrednictwem interfejsu `DataSource`; wiemy także, że podobne mechanizmy zarządzania pulą połączeń na poziomie kontenera (będące częścią szerszych rozwiązań w zakresie zarządzania transakcjami) są dostępne w większości środowisk kontenerowych J2EE, gdzie także mają postać implementacji interfejsu `DataSource`.

Warto jednak pamiętać, że angażowanie interfejsu DataSource w proces uzyskiwania połączeń z bazą danych wprowadza dodatkową złożoność, ponieważ tworzenie i uzyskiwanie samego egzemplarza implementacji tego interfejsu może się odbywać na wiele różnych sposobów. Przykładowo, implementacja DataSource dla Commons DBCP ma postać prostego komponentu JavaBean wypełnianego kilkoma właściwościami konfiguracyjnymi, natomiast w większości środowisk kontenerów J2EE egzemplarz interfejsu DataSource uzyskuje się za pośrednictwem interfejsu JNDI i wykorzystuje w ramach sekwencji kodu podobnej do tej, którą przedstawiono poniżej:

```
try {
  InitialContext context = new InitialContext();
  DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/datasource");
  // od tego miejsca możemy używać egzemplarza DataSource
  ...
}
catch (NamingException e) {
  // obsługuje wyjątek nazewnictwa w przypadku braku żądanego zasobu
}
```

Pozostałe implementacje DataSource mogą wymagać stosowania zupełnie innej strategii tworzenia i dostępu.

Być może nasz obiekt DAO mógłby sam wybierać właściwy sposób tworzenia lub uzyskiwania poszczególnych typów implementacji interfejsu DataSource — być może wybór odpowiedniej implementacji powinien być jednym z elementów konfiguracji. Ponieważ wiemy już co nieco o Springu i technice odwracania kontroli (IoC), doskonale zdajemy sobie sprawę, że nie byłoby to najlepsze rozwiązanie, ponieważ niepotrzebnie wiązałoby nasz obiekt DAO z implementacją DataSource i — tym samym — utrudniłoby konfigurację i bardzo skomplikowało proces testowania. Dość oczywistym rozwiązaniem jest uczynienie z implementacji interfejsu DataSource właściwości naszego DAO, którą będzie można wstrzykiwać do tego obiektu z poziomu kontenera Springa. Takie podejście doskonale zdaje egzamin w przypadku implementacji DataSource w wersji Commons DBCP, gdzie tworzony egzemplarz tego interfejsu jest w prosty sposób wstrzykiwany do obiektu DAO:

```
public class JdbcWeatherDaoImpl implements WeatherDao {

  DataSource dataSource;

  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public WeatherData find(Date date) {
    try {
      Connection con = dataSource.getConnection();
      // od tego miejsca możemy używać tego połączenia
    ...
  }
  ...
}
```

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClassName">
```

```
      <value>oracle.jdbc.driver.OracleDriver</value>
    </property>
    <property name="url">
      <value>jdbc:oracle:thin:@db-server:1521:devdb</value>
    </property>
    <property name="username"><value>john</value></property>
    <property name="password"><value>password</value></property>
  </bean>

  <bean id="weatherDao" class="ch02.sampleX.JdbcWeatherDaoImpl">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
  </bean>
```

Warto się jeszcze zastanowić nad sposobem wymiany wykorzystywanej implementacji interfejsu `DataSource` (w tym przypadku DBCP) na inną (np. uzyskiwaną ze środowiska JNDI). Wygląda na to, że rozwiązanie tego problemu nie będzie łatwe, ponieważ poza ustawieniem w naszym obiekcie DAO właściwości typu `DataSource` musimy otrzymać odpowiednią wartość JNDI — na razie wiemy tylko, jak przygotowywać konfigurację kontenera, aby definiowała właściwości JavaBean zawierające konkretne wartości lub odwołania do innych komponentów. Okazuje się, że realizacja tego zadania wcale nie wymaga wielkich umiejętności; wystarczy użyć klasy pomocniczej nazwanej `JndiObjectFactoryBean`:

```
  <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/datasource</value>
    </property>
  </bean>
```

```
  <bean id="weatherDao" class="ch02.sampleX.JdbcWeatherDaoImpl">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
  </bean>
```

Analiza komponentów fabrykujących

`JndiObjectFactoryBean` jest przykładem **komponentu fabrykującego**. Komponenty fabrykujące Springa bazują na bardzo prostej koncepcji — najprościej mówiąc, komponent fabrykujący to taki, który na żądanie generuje inny obiekt. Wszystkie komponenty fabrykujące Springa implementują interfejs `org.springframework.beans.factory.FactoryBean`. „Magia" tego rozwiązania tkwi we wprowadzeniu kolejnego poziomu pośredniczenia. Sam komponent fabrykujący jest tak naprawdę zwykłym komponentem JavaBean. Wdrażając komponent fabrykujący (tak jak każdy inny komponent JavaBean), musimy określić właściwości i argumenty konstruktora niezbędne do prawidłowego funkcjonowania tego komponentu. Okazuje się jednak, że kiedy inny komponent w ramach tego samego kontenera odwołuje się do tego komponentu fabrykującego (za pośrednictwem elementu `<ref>`) lub kiedy programista ręcznie zażądał tego komponentu (za pośrednictwem wywołania metody `getBean()`), kontener nie zwraca samego komponentu fabrykującego — wykrywa, że ma do czynienia z komponentem fabrykującym (za pomocą odpowiedniego interfejsu) i zamiast egzemplarza tego komponentu zwraca utworzony przez niego obiekt. Oznacza to, że z punktu

widzenia zależności każdy komponent fabrykujący w istocie może **być** traktowany jak obiekt, który jest przez niego zwracany. Przykładowo, w przypadku komponentu `JndiObjectFactoryBean` tym obiektem jest obiekt odnaleziony w środowisku JNDI (w naszym przykładzie egzemplarz implementacji interfejsu `DataSource`).

Interfejs `FactoryBean` jest bardzo prosty:

```
public interface FactoryBean {

  Object getObject() throws Exception;

  Class getObjectType();

  boolean isSingleton();

}
```

Wywoływana przez kontener metoda `getObject()` zwraca obiekt wynikowy. Metoda `isSingleton()` informuje, czy w kolejnych wywołaniach metody `getObject()` będzie zwracany ten sam egzemplarz obiektu czy wiele różnych egzemplarzy. I wreszcie metoda `getObjectType()` pozwala określić typ zwróconego obiektu (lub wartość `null`, jeśli typ tego obiektu nie jest znany). Komponenty fabrykujące są co prawda normalnie konfigurowane i wdrażane w kontenerach, jednak są także komponentami JavaBeans, zatem — w razie konieczności — można z nich korzystać także programowo.

> Warto się zastanowić, jak to możliwe, że komponent fabrykujący jest uzyskiwany za pośrednictwem wywołania metody `getBean()`, skoro odpowiedzią na żądanie pobrania komponentu fabrykującego jest jakiś inny obiekt wynikowy. Kluczem jest specjalny mechanizm, który sygnalizuje kontenerowi, że chodzi o sam komponent fabrykujący, a nie generowany przez niego obiekt wynikowy — wystarczy poprzedzić identyfikator komponentu znakiem `&`:
>
> ```
> FactoryBean facBean = (FactoryBean)appContext.getBean("&dataSource");
> ```

Tworzenie własnych, niestandardowych komponentów fabrykujących jest bardzo proste. Warto jednak pamiętać, że Spring zawiera wiele przydatnych implementacji tego typu komponentów, które oferują abstrakcje dostępu do większości tych popularnych zasobów i usług, w przypadku których stosowanie komponentów fabrykujących przynosi określone korzyści. Poniżej przedstawiono dość okrojoną listę tych implementacji:

- `JndiObjectFactoryBean` — zwraca obiekt będący wynikiem operacji wyszukiwania JNDI.
- `ProxyFactoryBean` — opakowuje istniejący obiekt wewnątrz odpowiedniego pośrednika, który jest następnie zwracany w odpowiedzi na żądanie. Rzeczywiste funkcjonowanie tego pośrednika zależy od konfiguracji zdefiniowanej przez użytkownika — może obejmować przechwytywanie i modyfikowanie zachowania obiektu, przeprowadzanie testów bezpieczeństwa itd. Techniki stosowania tego komponentu fabrykującego omówimy bardziej szczegółowo w rozdziale poświęconym frameworkowi programowania aspektowego (AOP) oferowanemu w ramach Springa.
- `TransactionProxyFactoryBean` — specjalizacja klasy `ProxyFactoryBean`, która opakowuje obiekt transakcyjnym pośrednikiem.

- `RmiProxyFactoryBean` — tworzy pośrednik obsługujący przezroczysty dostęp do zdalnego obiektu za pośrednictwem technologii zdalnego wywoływania metod (RMI). Podobne obiekty pośredniczące, tyle że dla dostępu do zdalnych obiektów przez protokoły HTTP, JAX-RPC, Hessian i Burlap są tworzone odpowiednio przez klasy `HttpInvokerProxyFactoryBean`, `JaxRpcPortProxyFactoryBean`, `HessianProxyFactoryBean` i `BurlapProxyFactoryBean`. We wszystkich przypadkach klient nie musi dysponować żadną wiedzą o pośredniku — wystarczy przystosowanie do współpracy z interfejsem biznesowym.
- `LocalSessionFactoryBean` — konfiguruje i zwraca obiekt `SessionFactory` frameworka Hibernate. Istnieją podobne klasy także dla innych mechanizmów zarządzania zasobami, w tym JDO i iBatis.
- `LocalStatelessSessionProxyFactoryBean` i `SimpleRemoteStatelessSessionProxyFactoryBean` — tworzą obiekt pośrednika wykorzystywany do uzyskiwania dostępu odpowiednio do lokalnych lub zdalnych bezstanowych komponentów sesyjnych EJB. Sam klient wykorzystuje tylko interfejs biznesowy (bez konieczności obsługi dostępu do JNDI lub interfejsów EJB).
- `MethodInvokingFactoryBean` — zwraca wynik wywołania metody innego komponentu, natomiast klasa `FieldRetrievingFactoryBean` zwraca wartość pola należącego do innego komponentu.
- Wiele komponentów fabrykujących związanych z technologią JMS zwraca jej zasoby.

Podsumowanie i efekt końcowy

W poprzednich punktach przekonaliśmy się, że konfigurowanie obiektów z wykorzystaniem technik IoC jest bardzo proste. Zanim jednak przystąpimy do wzajemnego wiązania obiektów, musimy te obiekty najpierw utworzyć lub uzyskać. W przypadku kilku potencjalnych obiektów współpracujących (nawet jeśli są ostatecznie udostępniane i konfigurowane za pośrednictwem standardowego interfejsu) sam fakt tworzenia i uzyskiwania rozmaitych obiektów za pomocą skomplikowanych i niestandardowych mechanizmów może stanowić przeszkodę w efektywnym przetwarzaniu obiektów. Okazuje się, że można tę przeszkodę wyeliminować za pomocą komponentów fabrykujących. Po początkowych fazach tworzenia i wiązania obiektów produkty generowane przez komponenty fabrykujące (osadzane w pośrednikach i innych podobnych obiektach opakowań) mogą pełnić funkcję adaptera — mogą pomagać w budowie abstrakcji ponad rzeczywistymi zasobami i usługami oraz zapewniać dostęp do zupełnie niepodobnych usług za pośrednictwem podobnych interfejsów.

Jak wiemy, źródła danych można dowolnie wymieniać bez konieczności umieszczania w kodzie klienta (w analizowanym przykładzie tę funkcję pełnił obiekt `weatherDao`) jakichkolwiek zapisów uzależniających od stosowanej technologii — przykładowo, zastąpiliśmy oryginalną implementację interfejsu `DataSource` opartą na Commons DBCP (mającą postać lokalnego komponentu) implementacją interfejsu `DataSource` właściwą dla technologii JNDI. Taka zamiana jest możliwa nie tylko dzięki mechanizmowi IoC, ale także dzięki wyprowadzeniu operacji dostępu do zasobów poza kod aplikacji i — tym samym — przygotowaniu właściwej struktury dla technik IoC.

> Mamy nadzieję, że dostrzegasz dwie najważniejsze zalety omawianego mechanizmu — z jednej możliwość łatwego abstrahowania od usług i zasobów, z drugiej strony możliwość wymiany jednego rozwiązania na inne bez konieczności wykorzystywania sposobów wdrożenia i technologii, których stosowanie podczas budowy wielu aplikacji najzwyczajniej w świecie mija się z celem. Cechą wyróżniającą Springa jest niezależność kodu klienta zarówno od faktycznego wdrożenia, jak i od użytego sposobu implementacji.

Skoro możemy w sposób transparentny, przezroczysty (przynajmniej z perspektywy klienta) uzyskiwać dostęp do zdalnych usług za pośrednictwem takich technologii jak RMI, RPC przez HTTP czy EJB, niby dlaczego nie mielibyśmy wdrażać całkowicie niezależnych rozwiązań, i po co w ogóle wiązać kod klienta z jedną implementacją ponad inną tam, gdzie nie jest to konieczne? W rozwiązaniach platformy J2EE można zaobserwować tendencję do udostępniania takich zasobów jak implementacje `DataSource`, zasoby JMS czy interfejsy JavaMail za pośrednictwem JNDI. Nawet jeśli ten sposób udostępniania zasobów znajduje jakieś uzasadnienie, bezpośredni dostęp klienta do środowiska JNDI jest całkowicie sprzeczny z założeniami efektywnego wytwarzania oprogramowania. Budowa dodatkowej warstwy abstrakcji np. za pomocą klasy `JndiObjectFactoryBean` oznacza, że w przyszłości będzie można zmienić środowisko na inne niż JNDI bez konieczności modyfikowania kodu klienta (odpowiednio dostosowując wyłącznie konfigurację samej fabryki komponentów). Nawet jeśli nie planujesz zmiany środowiska wdrożeniowego ani technologii implementacji już po wdrożeniu oprogramowania w docelowym środowisku, możesz być pewien, że taka abstrakcja znacznie ułatwi testy jednostkowe i testy integracyjne, ponieważ umożliwi weryfikację różnych konfiguracji wdrożeń i scenariuszy testowych. Tym zagadnieniom poświęcimy więcej miejsca w następnym rozdziale. Warto też podkreślić, że stosując komponent `JndiObjectFactoryBean`, eliminujemy potrzebę pisania zaawansowanego kodu w obiekcie DAO (konkretnie wyszukiwania obiektu w środowisku JNDI), który nie miałby nic wspólnego z właściwą funkcjonalnością biznesową tego obiektu. Ten przykład dobrze pokazuje, jak stosowanie techniki wstrzykiwania zależności ogranicza złożoność i nieład w kodzie źródłowym aplikacji.

Wielokrotne wykorzystywanie tych samych definicji komponentów

W porównaniu z ilością kodu Javy niezbędnego do pełnej konfiguracji i deklaracji wszystkich zależności definicje komponentów wyrażane w języku XML wydają się dość zwięzłe. Nie można jednak wykluczyć, że będziemy zmuszeni do opracowania wielu definicji komponentów, które w zdecydowanej większości będą do siebie bardzo podobne.

Przeanalizuj następującą definicję komponentu `WeatherService`:

```
<bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>
```

W typowym scenariuszu budowy aplikacji, gdzie nieodłącznym elementem zaplecza jest relacyjna baza danych, musielibyśmy korzystać z podobnych obiektów usług w sposób bardziej „transakcyjny". Szczegółowe omówienie tego zagadnienia znajdziesz w rozdziale 6.

Jednym z najprostszych sposobów osiągnięcia tego celu jest deklaratywne opakowanie takiego obiektu, aby nabrał cech obiektu transakcyjnego (np. za pomocą komponentu fabrykującego Springa nazwanego `TransactionProxyFactoryBean`):

```
<bean id="weatherServiceTarget" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>

<!-- pośrednik transakcyjny -->
<bean id="weatherService"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

  <property name="target"><ref local="weatherServiceTarget"/></property>

  <property name="transactionManager"><ref local="transactionManager"/></property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Jeśli na tym etapie nie do końca rozumiesz, jak należy szczegółowo konfigurować komponent `TransactionProxyFactoryBean` lub jak ten komponent faktycznie działa, nie masz powodów do zmartwień. Istotne jest to, że wspomniany komponent jest implementacją interfejsu `FactoryBean`, która dla otrzymanego na wejściu komponentu docelowego generuje obiekt transakcyjnego pośrednika (z jednej strony implementujący te same interfejsy co komponent docelowy, z drugiej strony obsługujący dodatkową semantykę przetwarzania transakcyjnego). Ponieważ chcemy, aby kod kliencki wykorzystywał obiekt opakowania, nazwa oryginalnego (nieopakowanego i nietransakcyjnego) komponentu jest zmieniana (rozszerzana o sufiks `Target`), a jego pierwotną nazwę otrzymuje wygenerowany komponent transakcyjny (w tym przypadku będą to odpowiednio `weatherServiceTarget` i `weatherService`). Oznacza to, że istniejący kod kliencki, który korzysta z usługi pogodowej, w ogóle „nie wie”, że od momentu zmiany konfiguracji korzysta z usługi transakcyjnej.

Opisany mechanizm deklaratywnego opakowywania obiektów jest co prawda bardzo wygodny (szczególnie w zestawieniu z rozwiązaniem alternatywnym, czyli działaniami programowymi na poziomie kodu), jednak w przypadku większych aplikacji z dziesiątkami lub setkami interfejsów usług, które wymagają stosowania bardzo podobnych technika opakowywania, wielokrotne definiowanie niemal identycznego, szablonowego kodu XML wydaje się stratą czasu. Okazuje się, że wprost idealnym rozwiązaniem tego problemu jest oferowana przez kontener obsługa definicji zarówno **macierzystych**, jak i **potomnych** komponentów. Korzystając z dobrodziejstw tego mechanizmu, możemy w prosty sposób zdefiniować abstrakcyjny, macierzysty (**szablonowy**) pośrednik transakcyjny:

```
<bean id="txProxyTemplate" abstract="true"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
<property name="transactionManager">
    <ref local="transactionManager"/>
  </property>
  <property name="transactionAttributes">
```

```
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Następnym krokiem jest utworzenie właściwych pośredników za pomocą odpowiednich definicji potomnych, które będą obejmowały tylko właściwości odróżniające tych pośredników od ich wspólnego, macierzystego szablonu (w tym przypadku komponentu `txProxyTemplate`):

```
<bean id="weatherServiceTarget" class="ch02.sample2.WeatherServiceImpl">
  <property name="weatherDao">
    <ref local="weatherDao"/>
  </property>
</bean>

<bean id="weatherService" parent="txProxyTemplate">
<property name="target"><ref local="weatherServiceTarget"/></property>
</bean>
```

Istnieje możliwość skorzystania z jeszcze bardziej klarownej i zwięzłej formy dziedziczenia. Ponieważ żaden z klientów nigdy nie będzie potrzebował dostępu do nieopakowanego komponentu naszej usługi pogodowej, można go zdefiniować jako komponent wewnętrzny względem pośrednika, który go opakowuje:

```
<bean id="weatherService" parent="txProxyTemplate">
<property name="target">
    <bean class="ch02.sample2.WeatherServiceImpl">
      <property name="weatherDao">
        <ref local="weatherDao"/>
      </property>
    </bean>
  </property>
</bean>
```

Jeśli jakaś inna usługa będzie wymagała podobnego opakowania, programista będzie mógł użyć definicji komponentu, która w bardzo podobny sposób odziedziczy właściwości macierzystego szablonu. W poniższym przykładzie odziedziczona właściwość `transactionAttributes` jest dodatkowo przykrywana — w ten sposób wprowadzono nowe (właściwe dla danego pośrednika) ustawienia propagowania transakcji:

```
<bean id="anotherWeatherService" parent="txProxyTemplate">
<property name="target">
    <bean class="ch02.sampleX.AnotherWeatherServiceImpl"/>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

Jeszcze prostszym rozwiązaniem jest skorzystanie z mechanizmu automatycznego pośredniczenia (ang. *autoproxying*) programowania aspektowego, który automatycznie wykrywa wszelkie cechy wspólne porad AOP w definicjach różnych komponentów. Więcej informacji na ten temat znajdziesz w rozdziale 4.

Cechy charakterystyczne definicji komponentów potomnych

Definicja komponentu potomnego dziedziczy wartości właściwości i argumenty konstruktora zadeklarowane w definicji komponentu macierzystego (przodka, rodzica). Okazuje się, że komponent potomny dziedziczy po definicji komponentu macierzystego także wiele opcjonalnych atrybutów (oczywiście pod warunkiem, że zadeklarowano je w definicji macierzystej). Zapewne zauważyłeś w poprzednim przykładzie, że do wskazywania identyfikatora komponentu macierzystego w definicji komponentu potomnego służy atrybut `parent`.

Definicja komponentu potomnego może wskazywać konkretną klasę lub pozostawiać odpowiedni atrybut nieustawiony i — tym samym — odziedziczyć jego wartość po definicji komponentu macierzystego. Warto pamiętać, że jeśli definicja potomka określa własną klasę (inną niż definicja komponentu macierzystego), nowa klasa musi prawidłowo obsługiwać wszystkie argumenty konstruktora i (lub) właściwości zadeklarowane w definicji rodzica, ponieważ zostaną one odziedziczone i użyte.

Potomek dziedziczy co prawda po swoim komponencie macierzystym wartości argumentów konstruktora, wartości właściwości i metody, jednak — w razie potrzeby — może dodawać także nowe, własne wartości. Z drugiej strony, dziedziczone przez komponent potomny wartości atrybutów `init-method`, `destroy-method` lub `factory-method` są w całości przykrywane przez odpowiednie wartości tego potomka.

Niektóre ustawienia konfiguracyjne komponentu macierzystego nigdy nie są dziedziczone i zawsze są określane w oparciu o definicję bieżącego (potomnego) komponentu. Należą do nich wartości atrybutów `depends-on`, `autowire`, `dependency-check`, `singleton` oraz `lazy-init`.

Egzemplarze komponentów oznaczonych jako abstrakcyjne (za pomocą atrybutu `abstract` — patrz powyższy przykład) jako takie nie mogą być tworzone. Jeśli nie przewidujesz sytuacji, w której będzie konieczne stworzenie egzemplarza komponentu macierzystego, zawsze powinieneś go oznaczać jako komponent abstrakcyjny. Należy to traktować jak dobrą praktykę, ponieważ egzemplarze nieabstrakcyjnego komponentu macierzystego mogą być tworzone przez kontener nawet wtedy, gdy tego nie zażądamy ani nie użyjemy żadnej zależności wskazującej na ten komponent. Konteksty aplikacji (ale nie fabryki komponentów) domyślnie podejmują próby utworzenia wstępnego egzemplarza dla każdego nieabstrakcyjnego komponentu singletonowego.

Warto pamiętać, że nawet jeśli nie użyjemy atrybutu `abstract` wprost, definicje komponentów mogą być niejawnie interpretowane jako abstrakcyjne ze względu na brak klasy lub metody fabrykującej, czyli de facto brak informacji niezbędnych do utworzenia ich egzemplarzy. Każda próba utworzenia egzemplarza abstrakcyjnego komponentu spowoduje błąd (niezależnie od tego, czy podjęto ją na żądanie programisty czy w sposób niejawny przez któryś z mechanizmów kontenera).

Stosowanie postprocesorów do obsługi niestandardowych komponentów i kontenerów

Postprocesory komponentów są w istocie specjalnymi **obiektami nasłuchującymi** (ang. *listeners*), które można rejestrować (wprost lub niejawnie) w kontenerze i które otrzymują ze strony kontenera wywołania zwrotne dla każdego komponentu, dla którego ten kontener tworzy egzemplarz. Postprocesory fabryk komponentów są bardzo podobne do postprocesorów komponentów z tą różnicą, że otrzymują wywołania zwrotne w momencie utworzenia egzemplarza samego **kontenera**. W sytuacji, gdy konieczne jest regularne dostosowywanie do zmieniających się warunków konfiguracji komponentu, grupy komponentów lub całego kontenera, można to zadanie bardzo ułatwić, tworząc własny postprocesor lub korzystając z jednego z wielu istniejących postprocesorów dołączanych do Springa.

Postprocesory komponentów

Postprocesory komponentów implementują interfejs `BeanPostProcessor`:

```
public interface BeanPostProcessor {
  Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException;
  Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException;
}
```

Interfejs `BeanPostProcessor` jest rozszerzany przez interfejs `DestructionAwareBeanPostProcessor`:

```
public interface DestructionAwareBeanPostProcessor extends BeanPostProcessor {
  void postProcessBeforeDestruction(Object bean, String beanName)
      throws BeansException;
}
```

W przedstawionej w tym rozdziale tabeli zdarzeń składających się na cykl życia komponentów dość precyzyjnie wskazano punkty, w których następują poszczególne wywołania zwrotne. Spróbujmy teraz stworzyć postprocesor komponentów, który będzie wykorzystywał wywołanie zwrotne metody `postProcessAfterInitialization()` do wypisywania na konsoli nazwy każdego komponentu, dla którego w danym kontenerze utworzono egzemplarz (wywołanie będzie następowało bezpośrednio po zdarzeniu utworzenia egzemplarza):

```
public class BeanInitializationLogger implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException {
    return bean;
  }

  public Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException {

    System.out.println("Zainicjalizowano komponent '" + beanName + "'");
    return bean;
  }
}
```

Przedstawiony przykład nie jest zbyt skomplikowany — rzeczywiste postprocesory najczęściej przetwarzają i zmieniają egzemplarze komponentów, a informacje o rejestrowanych zdarzeniach są zapisywane w pliku dziennika, a nie na konsoli.

W kontekście aplikacji postprocesory komponentów są rozpoznawane i wykorzystywane przez kontener automatycznie, a ich wdrażanie przebiega dokładnie tak jak w przypadku wszystkich innych komponentów:

```
<beans>
  <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
    <property name="weatherDao">
      <ref local="weatherDao"/>
    </property>
  </bean>
  <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl"/>
  <bean id="beanInitLogger" class="ch02.sample8.BeanInitializationLogger"/>
</beans>
```

Kiedy skonfigurowany w ten sposób kontekst aplikacji zostanie załadowany, nasz postprocesor otrzyma dwa wywołania zwrotne i w odpowiedzi wygeneruje (wyświetli na konsoli) następujące dane wyjściowe:

```
Zainicjalizowano komponent 'weatherDao'
Zainicjalizowano komponent 'weatherService'
```

Stosowanie postprocesorów komponentów dla nawet stosunkowo prostych fabryk komponentów jest nieco bardziej skomplikowane niż w przypadku kontekstów aplikacji, ponieważ wymaga ręcznego rejestrowania (co jest zgodne z duchem fabryk komponentów, który przewiduje bardziej programowe podejście), zamiast samego zadeklarowania postprocesora w formie komponentu w pliku konfiguracyjnym XML:

```
XmlBeanFactory factory =
        new XmlBeanFactory(new ClassPathResource("ch02/sample8/beans.xml"));
BeanInitializationLogger logger = new BeanInitializationLogger();
factory.addBeanPostProcessor(logger);
// Ponieważ nasze komponenty są singletonami, będą podlegały fazie wstępnego tworzenia egzemplarzy
// (także wówczas postprocesor otrzyma odpowiednie wywołania zwrotne).
factory.preInstantiateSingletons();
```

Jak widać, do utworzenia wstępnych egzemplarzy komponentów singletonowych w danej fabryce użyto metody `BeanFactory.preInstantiateSingletons()`, ponieważ domyślnie tylko konteksty aplikacji tworzą wstępne egzemplarze singletonów. Tak czy inaczej postprocesor jest wywoływany w momencie, w którym egzemplarz danego komponentu faktycznie jest tworzony, niezależnie od tego, czy w ramach procesu przygotowywania wstępnych egzemplarzy czy w odpowiedzi na konkretne żądanie (jeśli wspomniany proces zostanie pominięty).

Postprocesory fabryk komponentów

Postprocesory fabryk komponentów implementują interfejs `BeanFactoryPostProcessor`:

```
public interface BeanFactoryPostProcessor {
  void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
          throws BeansException;
}
```

Poniżej przedstawiono przykład postprocesora fabryki komponentów, którego działanie sprowadza się do pobrania i wyświetlenia listy nazw wszystkich komponentów w danej fabryce:

```
public class AllBeansLister implements BeanFactoryPostProcessor {

  public void postProcessBeanFactory(ConfigurableListableBeanFactory factory)
      throws BeansException {

    System.out.println("Fabryka zawiera następujące komponenty:");
    String[] beanNames = factory.getBeanDefinitionNames();
    for (int i = 0; i < beanNames.length; ++i)
      System.out.println(beanNames[i]);
  }
}
```

Samo używanie postprocesorów fabryki komponentów nie różni się zbytnio od stosowania postprocesorów komponentów. Ich wdrażanie w kontekście aplikacji przebiega tak samo jak w przypadku wszystkich innych komponentów — postprocesory zostaną automatycznie wykryte i użyte przez kontekst aplikacji. Z drugiej strony, w przypadku zwykłej fabryki komponentów postprocesor musi być uruchomiony ręcznie:

```
XmlBeanFactory factory = new XmlBeanFactory(
        new ClassPathResource("ch02/sample8/beans.xml"));
AllBeansLister lister = new AllBeansLister();
lister.postProcessBeanFactory(factory);
```

Przeanalizujmy teraz kilka najbardziej przydatnych postprocesorów komponentów i fabryk komponentów oferowanych w ramach Springa.

Postprocesor PropertyPlaceholderConfigurer

Podczas wdrażania aplikacji bazujących na Springu często się okazuje, że większość elementów konfiguracji kontenera nie będzie modyfikowana w czasie wdrażania. Zmuszanie kogokolwiek do przeszukiwania skomplikowanych plików konfiguracyjnych tylko po to, by zmienić kilka wartości, które tego **rzeczywiście** wymagają, bywa bardzo niewygodne. Takie rozwiązanie stwarza też niebezpieczeństwo popełnienia przypadkowego błędu w pliku konfiguracyjnym, np. przez nieumyślne zmodyfikowanie niewłaściwej wartości.

`PropertyPlaceholderConfigurer` jest postprocesorem fabryki komponentów, który — użyty w definicji fabryki komponentów lub kontekstu aplikacji — umożliwia określanie pewnych wartości za pośrednictwem specjalnych **łańcuchów zastępczych**, **łańcuchów-wypełniaczy** (ang. *placeholder strings*), które są następnie zastępowane przez rzeczywiste wartości pochodzące

z pliku zewnętrznego (w formacie Java Properties). Co więcej, mechanizm konfiguracyjny domyślnie będzie weryfikował te łańcuchy pod kątem zgodności z właściwościami systemowymi Javy, jeśli nie znajdzie odpowiednich dopasowań w pliku zewnętrznym. Tryb takiej dodatkowej weryfikacji można włączać i wyłączać za pomocą właściwości `systemPropertiesMode` konfiguratora (która oferuje też możliwość wymuszania pierwszeństwa dopasowań do właściwości System Properties względem zapisów zewnętrznego pliku właściwości).

Dobrym przykładem wartości, które warto wyprowadzić poza skomplikowane pliki konfiguracyjne, są łańcuchy konfiguracyjne właściwe dla puli połączeń z bazą danych. Poniżej przedstawiono ponownie fragment kodu definiującego implementację `DataSource` korzystającą z Commons DBCP — tym razem użyto łańcuchów zastępczych zamiast rzeczywistych wartości:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClassName"><value>${db.driverClassName}</value></property>
  <property name="url"><value>${db.url}</value></property>
  <property name="username"><value>${db.username}</value></property>
  <property name="password"><value>${db.password}</value></property>
</bean>
```

Rzeczywiste wartości będą pochodziły z zewnętrznego pliku właściwości nazwanego *jdbc.properties*:

```
db.driverClassName=oracle.jdbc.driver.OracleDriver
db.url=jdbc:oracle:thin:@db-server:1521:devdb
db.username=john
db.password=password
```

Aby użyć egzemplarza klasy `PropertyPlaceholderConfigurer` i za jej pomocą wydobyć właściwe wartości z pliku właściwości do kontekstu aplikacji, wystarczy wdrożyć ten egzemplarz tak jak każdy inny komponent:

```
<bean id="placeholderConfig"
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location"><value>jdbc.properties</value></property>
</bean>
```

Aby użyć teraz zadeklarowanego przed chwilą komponentu konfiguratora dla prostej fabryki komponentów, należy go uruchomić ręcznie:

```
XmlBeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
PropertyPlaceholderConfigurer ppc = new PropertyPlaceholderConfigurer();
ppc.setLocation(new FileSystemResource("db.properties"));
ppc.postProcessBeanFactory(factory);
```

Postprocesor PropertyOverrideConfigurer

`PropertyOverrideConfigurer`, który także jest postprocesorem fabryk komponentów, co prawda pod wieloma względami przypomina postprocesor `PropertyPlaceholderConfigurer`, jednak o ile w przypadku tego drugiego wartości **musiały** pochodzić z zewnętrznego pliku

właściwości, o tyle `PropertyOverrideConfigurer` umożliwia **przykrywanie** tymi wartościami wartości właściwości komponentów w fabryce komponentów lub kontekście aplikacji.

Każdy wiersz pliku właściwości musi mieć następujący format:

```
beanName.property=value
```

gdzie `beanName` reprezentuje identyfikator komponentu, `property` jest jedną z właściwości tego komponentu, a `value` reprezentuje docelową wartość tej właściwości. Plik właściwości może zawierać np. następujące zapisy:

```
dataSource.driverClassName=oracle.jdbc.driver.OracleDriver
dataSource.url=jdbc:oracle:thin:@db-server:1521:devdb
dataSource.username=john
dataSource.password=password
```

W ten sposób można przykryć cztery właściwości komponentu `dataSource`. Wszelkie właściwości (wszystkich komponentów) w danym kontenerze, które nie zostały przykryte przez nowe wartości przedstawionego pliku właściwości, utrzymają swoje dotychczasowe wartości — albo te zdefiniowane w konfiguracji kontenera, albo wartości domyślne komponentu (w przypadku braku odpowiednich zapisów w pliku konfiguracyjnym). Ponieważ sama analiza konfiguracji kontenera nie pozwala określić, czy dana wartość zostanie przykryta (dla pewności konieczne jest jeszcze przestudiowanie pliku właściwości), tego typu funkcjonalność należy wykorzystywać tylko w uzasadnionych przypadkach.

Postprocesor CustomEditorConfigurer

`CustomEditorConfigurer` jest postprocesorem fabryki komponentów, za pomocą którego możemy rejestrować własne edytory właściwości (rozszerzenia klasy `PropertyEditorSupport`) komponentów JavaBeans obsługujące konwersję wartości łańcuchowych na docelowe wartości właściwości lub argumentów konstruktora (w konkretnym, często złożonym formacie obiektowym).

Tworzenie własnych edytorów właściwości

Częstym powodem stosowania własnych, niestandardowych edytorów właściwości jest konieczność ustawiania właściwości typu `java.util.Date`, których wartości źródłowe zapisano w postaci łańcuchów. Ponieważ formaty dat bardzo często są uzależnione od ustawień regionalnych, użycie egzemplarza `PropertyEditor` eksportującego określony format łańcucha źródłowego jest najprostszym sposobem radzenia sobie z problemem niezgodności formatów. Ponieważ utrudnienia związane z formatami dat dotyczą większości użytkowników, zamiast prezentować jakiś abstrakcyjny przykład, posłużymy się klasą `CustomDateEditor` (implementacją `PropertyEditor` dla typu `Date`) — przeanalizujemy nie tylko kod tej klasy, ale także techniki jej rejestrowania i wykorzystywania w praktyce. Własne edytory właściwości powinny być implementowane i rejestrowane w bardzo podobny sposób:

```
public class CustomDateEditor extends PropertyEditorSupport {

  private final DateFormat dateFormat;
  private final boolean allowEmpty;
```

```
    public CustomDateEditor(DateFormat dateFormat, boolean allowEmpty) {
      this.dateFormat = dateFormat;
      this.allowEmpty = allowEmpty;
    }

    public void setAsText(String text) throws IllegalArgumentException {
      if (this.allowEmpty && !StringUtils.hasText(text)) {
        // traktuje pusty łańcuch jak wartość null
        setValue(null);
      }
      else {
        try {
          setValue(this.dateFormat.parse(text));
        }
        catch (ParseException ex) {
          throw new IllegalArgumentException("Konwersja daty nie powiodła się: " +
                  ex.getMessage());
        }
      }
    }

    public String getAsText() {
      return (getValue() == null ? "" : this.dateFormat.format((Date)getValue()));
    }
  }
```

Pełna dokumentacja tej klasy w formacie JavaDoc jest dołączana do wersji instalacyjnej Springa. Implementację własnego edytora właściwości najlepiej rozpocząć od rozszerzenia pomocniczej klasy bazowej `java.beans.PropertyEditorSupport` będącej częścią standardowej biblioteki klas Javy. Wspomniana klasa implementuje większość potrzebnych mechanizmów edycji właściwości (także poza standardowymi metodami `setAsText()` i `getAsText()`, których implementacje zawsze musimy przykrywać). Warto pamiętać, że chociaż egzemplarze `PropertyEditor` mają swój stan i w normalnych warunkach nie gwarantują bezpieczeństwa przetwarzania wielowątkowego, sam Spring zapewnia odpowiednie mechanizmy synchronizacji całej sekwencji wywołań metod niezbędnych do prawidłowego przeprowadzania konwersji.

`CustomDateEditor` może korzystać z dowolnej (przekazywanej za pośrednictwem argumentu jej konstruktora) implementacji interfejsu `java.text.DateFormat` do przeprowadzania właściwej implementacji. Wdrażając klasę `CustomDateEditor`, możesz użyć implementacji `java.text.SimpleDateFormat`. Nasz edytor właściwości można też skonfigurować w taki sposób, aby łańcuchy puste albo interpretował jak wartości `null`, albo traktował jak błąd nieprawidłowego argumentu.

Rejestrowanie i używanie własnych, niestandardowych edytorów właściwości

Przyjrzyjmy się teraz definicji kontekstu aplikacji, w której użyto postprocesora `CustomEditorConfigurer` do zarejestrowania edytora `CustomDateEditor` (implementacji uniwersalnego edytora `PropertyEditor`), którego zadaniem jest konwersja łańcuchów znakowych na obiekty klasy `java.util.Date`. Definicja konfiguracji zawiera konkretny format łańcuchów reprezentujących daty:

```
<bean id="customEditorConfigurer"
      class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>

      <!-- rejestruje edytor właściwości dla typu java.util.Date -->
      <entry key="java.util.Date">
        <bean class="org.springframework.beans.propertyeditors.CustomDateEditor">
          <constructor-arg index="0">
            <bean class="java.text.SimpleDateFormat">
              <constructor-arg><value>M/d/yy</value></constructor-arg>
            </bean>
          </constructor-arg>
          <constructor-arg index="1"><value>true</value></constructor-arg>
        </bean>
      </entry>

    </map>
  </property>
</bean>

<!-- sprawdza funkcjonowanie edytora dat, ustawiając dwie właściwości typu Date w postaci łańcuchów -->
<bean id="testBean" class="ch02.sample9.StartEndDatesBean">
  <property name="startDate"><value>10/09/1968</value></property>
  <property name="endDate"><value>10/26/2004</value></property>
</bean>
```

Postprocesor `CustomEditorConfigurer` może rejestrować jeden lub wiele niestandardowych edytorów właściwości (choć w prezentowanym przykładzie ograniczymy się do jednego, `CustomDateEditor`). Twoje edytory (zaimplementowane z myślą o innych typach) mogą nie wymagać żadnych specjalnych zabiegów konfiguracyjnych. Z powyższego kodu konfiguracji wynika, że klasa `CustomDateEditor`, której konstruktor pobiera na wejściu dwa argumenty, otrzyma obiekt klasy `SimpleDateFormat` reprezentujący łańcuch formatu daty oraz wartość logiczną (typu `boolean`) określającą, czy łańcuchy puste powinny być traktowane jak wartości `null`.

Przedstawiony przykład ilustruje też wygodny sposób definiowania komponentu testowego (w tym przypadku nazwanego `testBean`), który zawiera dwie właściwości typu `Date` ustawiane za pośrednictwem wartości łańcuchowych — w ten sposób sprawdzamy, czy nasz edytor właściwości działa prawidłowo.

Postprocesor BeanNameAutoProxyCreator

`BeanNameAutoProxyCreator` jest postprocesorem komponentów. Co prawda techniki korzystania z tego postprocesora omówimy bardziej szczegółowo w rozdziale poświęconym programowaniu aspektowemu, jednak już teraz dobrze jest wiedzieć o jego istnieniu. Najkrócej mówiąc, dla danej na wejściu listy nazw komponentów postprocesor `BeanNameAutoProxyCreator` może opakować te spośród otrzymanych komponentów danej fabryki, których nazwy pasują do nazw znajdujących się na tej liście (proces opakowywania następuje w czasie tworzenia egzemplarzy komponentów, a zadaniem obiektów pośredniczących jest albo przechwytywanie operacji dostępu do oryginalnych komponentów, albo modyfikowanie ich zachowań).

Postprocesor DefaultAdvisorAutoProxyCreator

Ten postprocesor komponentów przypomina opisany przed chwilą postprocesor `BeanNameAutoProxyCreator`, jednak oprócz samych nazw komponentów przeznaczonych do opakowania odnajduje też informacje na temat sposobu tego opakowania (tzw. porady). Także w tym przypadku warto się zapoznać z treścią rozdziału poświęconego technikom programowania aspektowego (AOP).

Podsumowanie

Po przeczytaniu tego rozdziału powinieneś znacznie lepiej rozumieć, co tak naprawdę oznaczają takie pojęcia jak **odwracanie kontroli** i **wstrzykiwanie zależności** oraz jak obie koncepcje zostały urzeczywistnione w postaci fabryk komponentów i kontekstów aplikacji Springa. Przeanalizowaliśmy i użyliśmy większości podstawowych elementów funkcjonalności kontenera Springa. Ponieważ właśnie kontener IoC jest podstawą pozostałych mechanizmów Springa, dobre rozumienie sposobu jego funkcjonowania i technik konfiguracji w połączeniu ze świadomością potencjału tego kontenera jest kluczem do efektywnego korzystania ze Springa.

Dowiedziałeś się między innymi:

- Jak dzięki funkcjom kontenera można korzystać z zalet jednego, logicznie spójnego i przewidywalnego mechanizmu dostępu, konfigurowania i wiązania obiektów (zamiast używać programowych lub tworzonych naprędce mechanizmów łączenia klas, które tylko utrudniają testowanie). Ogólnie, kontener pozwala całkowicie wyeliminować konieczność korzystania z niestandardowych, przystosowanych do konkretnych klas fabryk komponentów oraz singletonów.
- Jak kontener wspiera programistę w stosowaniu pożądanej praktyki oddzielania interfejsu od implementacji w kodzie aplikacji.
- Że postprocesory oferują możliwość elastycznego dostosowywania zachowania komponentów i kontenera (za pomocą odpowiednich plików zewnętrznych).
- Jakie są podstawowe reguły odwracania kontroli i jak w połączeniu z fabryką komponentów tworzą doskonałe narzędzie do budowy abstrakcji ponad operacjami wymagającymi dostępu do usług i zasobów.
- Że technika IoC wraz z odpowiednim kontenerem może stanowić solidną podstawę dla budowy aplikacji w oparciu o framework Spring bez konieczności tworzenia ścisłych związków pomiędzy konstruowanym oprogramowaniem a samym kontenerem.

W następnym rozdziale skupimy się na bardziej zaawansowanych mechanizmach kontekstu aplikacji i przedstawimy kilka zaawansowanych scenariuszy użycia kontenera.