IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

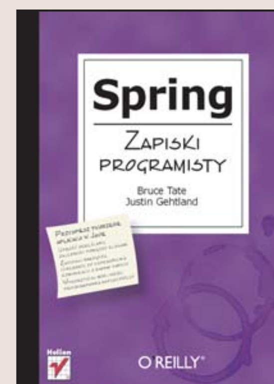
ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Spring. Zapiski programisty



Autorzy: Bruce A. Tate, Justin Gehtland
Tłumaczenie: Piotr Rajca
ISBN: 83-246-0336-0
Tytuł oryginału: Spring: A Developers Notebook
Format: B5, stron: 272

Przyspiesz tworzenie aplikacji w Javie

- Uprość określanie zależności pomiędzy klasami
- Zastosuj narzędzie Hibernate do usprawnienia komunikacji z bazami danych
- Wykorzystaj możliwości programowania aspektowego

Technologia J2EE miała w założeniu być prosta i szybka w użytkowaniu. Praktyka jednak okazała się daleka od teorii. Mozolne tworzenie aplikacji wykorzystujących dziesiątki interfejsów połączonych wzajemnymi zależnościami, setki deskryptorów wdrożenia oraz plików pomocniczych spowodowało, że zaczęto poszukiwać rozwiązań alternatywnych. Jednym z nich okazał się framework Spring, zyskujący coraz większą popularność wśród programistów Javy. Spring jest znacznie dużo prostszy od wielu alternatywnych rozwiązań J2EE, znacznie ułatwia testowanie aplikacji, i pozwala na usuwanie zależności z kodu oraz oddzielanie ich od serwera aplikacji. Spring umożliwia również wykorzystanie programowania aspektowego.

Książka „Spring. Zapiski programisty" to praktyczny przewodnik po możliwościach tego środowiska. Jeśli wolisz poznawać nowe zagadnienia w sposób praktyczny, a nie wertując setki stron zapełnionych teoretycznymi wywodami, to ta książka jest właśnie dla Ciebie. Znajdziesz w niej omówienie zagadnień związanych z samym Springiem, współpracującymi z nim narzędziami i sposobami wykorzystania ich w procesie tworzenia aplikacji J2EE — począwszy do graficznego interfejsu użytkownika i interfejsu sieciowego, a skończywszy na dostępie do relacyjnych baz danych.

- Tworzenie klas z zastosowaniem zależności
- Budowanie interfejsu użytkownika
- Integrowanie JSF z frameworkiem Spring
- Dostęp do baz danych za pomocą JDBC
- Odwzorowanie baz danych na obiekty za pomocą Hibernaete
- Obsługa i zabezpieczanie transakcji
- Wysyłanie i odbieranie wiadomości e-mail

Jeśli poszukujesz wydajniejszych metod tworzenia aplikacji J2EE, wykorzystaj możliwości frameworka Spring. Dzięki tej książce poznasz je wszystkie.

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

Spis treści

Przedmowa **5**

Wstęp **9**

Rozdział 1. Początki **17**

Tworzenie dwóch klas przy wykorzystaniu zależności 18
Stosowanie wstrzykiwania zależności 23
Automatyzacja przykładu 28
Wstrzykiwanie zależności przy wykorzystaniu frameworka Spring 32
Tworzenie testu 35

Rozdział 2. Tworzenie interfejsu użytkownika **41**

Konfiguracja Tomcata 42
Tworzenie widoku przy wykorzystaniu Web MVC 46
Wzbogacanie aplikacji sieciowych 56
Testowanie 65

Rozdział 3. Integracja innych klientów **69**

Tworzenie interfejsu użytkownika w oparciu o framework Struts 70
Stosowanie JSF wraz z frameworkiem Spring 83
Integracja JSF z frameworkiem Spring 92

Rozdział 4. Stosowanie JDBC **95**

Konfiguracja bazy danych i utworzenie schematu 96
Stosowanie szablonów JDBC frameworka Spring 101
Wydzielanie często używanego kodu 108
Stosowanie obiektów dostępowych 110
Wykonywanie testów przy użyciu szkieletu EasyMock 116

Rozdział 5. Odwzorowania obiektowo-relacyjne 121

Integracja frameworka iBATIS 123

Stosowanie frameworka Spring z JDO 134

Stosowanie frameworków Hibernate oraz Spring 142

Testowanie 150

Rozdział 6. Usługi i AOP 151

Tworzenie usługi 152

Konfiguracja usługi 159

Stosowanie automatycznych obiektów pośredniczących 164

Porady operujące na wyjątkach 167

Testowanie usługi przy wykorzystaniu obiektów zastępczych 170

Testowanie usługi mającej efekty uboczne 174

Rozdział 7. Transakcje i bezpieczeństwo 177

Programowa obsługa transakcji 178

Konfiguracja prostych transakcji 182

Transakcje obejmujące kilka baz danych 184

Zabezpieczenie serwletów aplikacji 190

Zabezpieczanie metod aplikacji 200

Tworzenie obiektu przechwytującego ułatwiającego testowanie 206

Rozdział 8. Obsługa wiadomości i praca zdalna 211

Wysyłanie wiadomości poczty elektronicznej 212

Praca zdalna 216

Stosowanie JMS 219

Testowanie aplikacji JMS 224

Rozdział 9. Tworzenie grubych klientów 229

Zaczynamy pracę 229

Tworzenie widoku BikeNavigator 244

Tworzenie formularzy edytora rowerów 249

Skorowidz 259

ROZDZIAŁ 1.

Początki

W tym rozdziale:

- ✓ Tworzenie dwóch klas przy wykorzystaniu zależności
- ✓ Stosowanie wstrzykiwania zależności
- ✓ Automatyzacja przykładu
- ✓ Wstrzykiwanie zależności przy wykorzystaniu frameworka Spring
- ✓ Tworzenie testu

W weekendy z wielkim zapałem oddaję się sportom ekstremalnym. Według standardów właściwych dla osób przeciętnych, robiłem na rowerze górskim naprawdę zwariowane rzeczy, a oprócz tego uwielbiam kajakarstwo górskie. Tylko raz znalazłem się w poważnych kłopotach. Spływałem na rzece o nazwie Piedra, na której byłem po raz pierwszy, w nowym sprzęcie i z głową naładowaną nowymi technikami. W kajakarstwie górskim przeżycie zależy od zachowania prostoty i szybkości, a ja byłem niepewny i wolny. Zamiast atakować rzekę, to ona zaatakowała mnie; przepłynąłem przez trzy bystrza klasy IV i naprawdę miałem szczęście, że wyszedłem z tego bez żadnych obrażeń.

Szybkość i prostota ma równie duże znaczenie w przypadku tworzenia oprogramowania. *Spring Framework* daje mi jedno i drugie. Kupiłeś tę książkę, więc zapewne zgadzasz się z tym stwierdzeniem. Spring, choć jest prosty, ma bardzo duże możliwości. Ogromne. Spring pozwoli Ci na pisanie aplikacji o architekturze warstwowej, w której poszczególne warstwy

będę wyraźnie od siebie oddzielone. Spring sprawia, że będziesz mógł testować tworzony kod w tak prosty i przejrzysty sposób, o jakim wcześniej nawet nie mogłeś marzyć. W niniejszym rozdziale stworzysz prostą aplikację, zautomatyzujesz proces jej kompilacji i przystosujesz ją do korzystania z frameworka Spring.

Spring jest najpopularniejszym produktem z nowej grupy tak zwanych lekkich kontenerów. Jeśli uwzględnimy wszystkie wchodzące w jego skład moduły, to się okaże, że Spring nie jest aż taki „lekki", jednak w tym przypadku określenie to odnosi się do tego wszystkiego, co zazwyczaj trzeba dodawać do kodu umieszczanego i wykonywanego w kontenerze.

Tworzenie dwóch klas przy wykorzystaniu zależności

Wielu nauczycieli i konsultantów pisze o zależnościach jako o czymś, co można by porównać do majonezu zbyt długo pozostawionego na słońcu. Jednak jeśli tworzona aplikacja ma robić cokolwiek interesującego, to zależności muszą się w niej pojawić. Cała sztuka polega na tym, by zidentyfikować ważne zależności i obsłużyć je w odpowiedni sposób. Sposób, w jaki rozwiążesz zależności, będzie mieć znaczący wpływ na łatwość utrzymania aplikacji, jej rozbudowy oraz testowania. Podczas lektury niniejszej książki napiszesz system rezerwacji rowerów górskich. Sklep sportowy mógłby korzystać z takiego systemu do rezerwacji i wypożyczania rowerów. Zaczniemy od najprostszych rozwiązań, w których wszystkie zależności będą określone na stałe w kodzie; w ten sposób będziemy mogli się upewnić, że używana infrastruktura działa poprawnie. Następnie rozluźnimy powiązania za pomocą frameworka Spring i stopniowo będziemy dodawać do naszej aplikacji trwałość, warstwę prezentacji sieciowej, deklaratywne transakcje i kilka innych usług.

Dla mnie tworzenie przy wykorzystaniu frameworka Spring ma charakter iteracyjny. Najbliższych kilka ćwiczeń jest poświęconych powolnemu przygotowywaniu infrastruktury. W pierwszym upewnimy się, że dysponujemy poprawnie skonfigurowanym oraz działającym środowiskiem Javy i zaczniemy tworzyć podstawowy model aplikacji. Każdy z kilku pierwszych przykładów jest poświęcony niewielkiemu fragmentowi środowiska, co ułatwi nam rozwiązywanie ewentualnych problemów, gdyby jakieś się pojawiły.

Jak to osiągnąć?

Zaczniemy od dwóch klas umieszczonych w jednym katalogu. Pierwsza z nich będzie reprezentować rower górski, a druga rejestr, w którym będą przechowywane informacje o wszystkich rowerach. Tę drugą klasę nazwiemy fasadą. Informacje o posiadanych rowerach będą podawane w konstruktorze fasady, a następnie wyświetlane przy wykorzystaniu trzeciej, bardzo prostej klasy.

Klasa `Bike`, reprezentująca rower, została przedstawiona na listingu 1.1.

Listing 1.1. Bike.java

```
public class Bike {
   private String manufacturer;
   private String model;
   private int frame;
   private String serialNo;
   private double weight;
   private String status;

   public Bike() {}

   public Bike(String manufacturer, String model, int frame, String
   serialNo, double weight, String status) {
      this.manufacturer = manufacturer;
      this.model = model;
      this.frame = frame;
      this.serialNo = serialNo;
      this.weight = weight;
      this.status = status;
   }

   public String toString() {
      return "Rower : " +
            "producent -- " + manufacturer +
            "\n: model -- " + model +
            "\n: rama -- " + frame +
            "\n: numer seryjny -- " + serialNo +
            "\n: waga -- " + weight +
            "\n: status -- " + status +
            ".\n"; }

   public String getManufacturer() { return manufacturer; }

   public void setManufacturer(String manufacturer) {
      this.manufacturer = manufacturer;
   }
```

```
    public String getModel() { return model; }

    public void setModel(String model) { this.model = model; }

    public int getFrame() { return frame; }

    public void setFrame(int frame) { this.frame = frame; }

    public String getSerialNo() { return serialNo; }

    public void setSerialNo(String serialNo) { this.serialNo = serialNo; }

    public double getWeight() { return weight; }

    public void setWeight(double weight) { this.weight = weight; }

    public String getStatus() { return status; }

    public void setStatus(String status) { this.status = status; }
}
```

Listing 1.2 przedstawia fasadę.

Listing 1.2. RentABike.java

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class RentABike {

    private String storeName;
    final List bikes = new ArrayList();

    public RentABike(String storeName) {
        this.storeName = storeName;
        bikes.add(new Bike("Shimano", "Roadmaster", 20, "11111", 15,
                "Dobry"));
        bikes.add(new Bike("Cannondale", "F2000 XTR", 18, "22222",12,
                "Doskonały"));
        bikes.add(new Bike("Trek","6000", 19, "33333", 12.4,
                "Dobry"));
    }

    public String toString() { return "RentABike: " + storeName; }

    public List getBikes() { return bikes; }

    public Bike getBike(String serialNo) {
        Iterator iter = bikes.iterator();
        while(iter.hasNext()) {
            Bike bike = (Bike)iter.next();
            if(serialNo.equals(bike.getSerialNo())) return bike;
```

```
      }
      return null;
   }
}
```

I w końcu kod zamieszczony na listingu 1.3 przedstawia widok.

Listing 1.3. CommandLineView.java

```
import java.util.Iterator;

public class CommandLineView {
   private RentABike rentaBike;
   public CommandLineView() {rentaBike = new RentABike("Rowery Bruce'a");}

   public void printAllBikes() {
      System.out.println(rentaBike.toString());
      Iterator iter = rentaBike.getBikes().iterator();
      while(iter.hasNext()) {
         Bike bike = (Bike)iter.next();
         System.out.println(bike.toString());
      }
   }

   public static void main(String[] args) {
      CommandLineView clv = new CommandLineView();
      clv.printAllBikes();
   }
}
```

Klasa RentABike jest zależnością. W przypadku stosowania takiego stylu programowania zależności są podawane na stałe, co zwiększa stopień wzajemnego powiązania fasady i widoku. Framework Spring pomoże nam wyeliminować zależności takiego typu.

Teraz możesz już skompilować aplikację, używając następującego polecenia:

```
C:\PozyczRowerApp\src javac -d ..\out *.java
```

Katalog wynikowy będzie teraz zawierać skompilowane pliki klas:

```
Katalog: C:\PozyczRowerApp\out

2005-12-27  08:17    <DIR>          .
2005-12-27  08:17    <DIR>          ..
2005-12-27  08:17             1 753 Bike.class
2005-12-27  08:17             1 442 RentABike.class
2005-12-27  08:17               937 CommandLineView.class
```

Aplikację możesz uruchomić w następujący sposób:

```
C:\PozyczRowerApp\out> java CommandLineView

RentABike: Rowery Bruce'a
Rower : producent -- Shimano
: model -- Roadmaster
: rama -- 20
: numer seryjny -- 11111
: waga -- 15.0
: status -- Dobry.
```

```
Rower : producent -- Cannondale
: model -- F2000 XTR
: rama -- 18
: numer seryjny -- 22222
: waga -- 12.0
: status -- Doskonały.

Rower : producent -- Trek
: model -- 6000
: rama -- 19
: numer seryjny -- 33333
: waga -- 12.4
: status -- Dobry.
```

Jak to działa?

Nic dobrego. Nasz aktualny projekt jest bardzo prosty, lecz jednocześnie wszystkie powiązania są określone na stałe. Od razu można wskazać kilka wad takiego rozwiązania:

Podstawowy wzorzec projektowy stosowany we frameworku Spring oraz innych lekkich kontenerach, bazuje na zmniejszeniu stopnia powiązań pomiędzy zależnościami.

- Warstwa fasady (`RentABike`) w statyczny sposób tworzy rowery dostępne w sklepie, a zatem, za każdym razem, gdy pojawi się jakiś nowy rower (ech!), bądź też gdy użytkownik „skasuje" jakiś rower (co za pech!), konieczne będzie wprowadzanie odpowiednich zmian w kodzie.
- Także przetestowanie modelu będzie kłopotliwe, gdyż zbiór rowerów jest stały i niezmienny.
- Interfejs użytkownika oraz fasada są ze sobą ściśle powiązane. Można wskazać podaną na stałe zależność pomiędzy warstwą fasady i interfejsem użytkownika.
- Kod źródłowy nie jest w żaden sposób zorganizowany, a proces kompilacji aplikacji nie jest zautomatyzowany.

Jednak teraz chcemy wprowadzić pewne ułatwienia, by poprawić działanie środowiska Javy, tworząc jednocześnie podwaliny naszej aplikacji. W następnym przykładzie rozdzielimy poszczególne warstwy aplikacji i zautomatyzujemy proces jej kompilacji.

A co z…

… przygotowaniem tego całego burrito raz a dobrze? Możesz zdecydować, by jednocześnie zainstalować Spring, Hiberanate, Javę, Ant oraz JUtil. Jednak z własnego doświadczenia mogę stwierdzić, że zajmując się kolejno każdym zagadnieniem z osobna, w rzeczywistości oszczędzamy czas; zwłaszcza jeśli tworzymy podstawy środowiska pracy. Kiedy wszystkie narzędzia zostaną już zainstalowane i będą działać poprawnie, będziesz mógł połączyć kilka czynności.

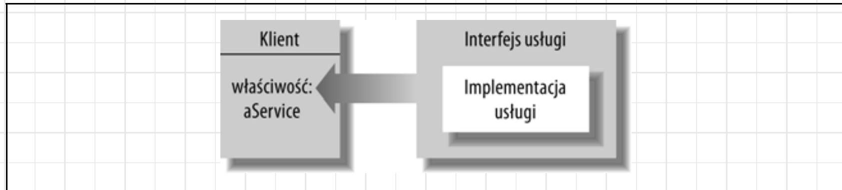
Stosowanie wstrzykiwania zależności

Kiedy zaczynam tworzyć aplikację, zazwyczaj jej poszczególne elementy są ze sobą ściśle powiązane. Nie ma w tym nic złego. W każdej chwili mogę to zmienić w późniejszych etapach pracy. W niniejszym przykładzie zmienię strukturę aplikacji w taki sposób, aby współpracowała z frameworkiem Spring w przypadku, gdy zostanie on wykorzystany. W tym celu dodam do warstwy fasady interfejs, aby mogła ona implementować kilka różnych strategii.

Wiele osób, gdy po raz pierwszy styka się z wstrzykiwaniem zależności, dziwi się: „O co tyle hałasu?”. Jednak kiedy zaczną używać tej techniki, po pewnym czasie zrozumieją, iż ta prosta zmiana może w ogromnym stopniu poprawić tworzony kod i ułatwić utrzymanie aplikacji.

Pierwszym krokiem podczas poznawania frameworka Spring jest poznanie wzorca *wstrzykiwania zależności* (ang. *dependency injection*). Nie jest on skomplikowany, choć ma kluczowe znaczenie. Jest on na tyle odmienny od standardowo przyjętego sposobu kodowania, że zapewne będziesz chciał zanotować sobie skrócone informacje na jego temat.

Rysunek 1.1 przedstawia klienta i serwer przygotowane do wykorzystania techniki wstrzykiwania zależności. Klient używa innej klasy, którą będziemy nazywali *usługą*. Klient posiada właściwość, a w niej można zapisać referencję do usługi. Usługa jest z kolei reprezentowana przez interfejs, co sprawia, że klient nie zna jej faktycznej implementacji. Niemniej jednak taki kod nie cechuje się niskim stopniem powiązań — wciąż w jakiś sposób musimy utworzyć usługę. Dzięki wstrzykiwaniu zależności jakieś dodatkowe narzędzie, nazywane *asemblerem* lub *kontenerem*, odpowiada za utworzenie zarówno klienta, jak i usługi, a następnie określa wartość właściwości `aService` (będącej referencją do obiektu typu `Service` stanowiącego implementację usługi), spełniając tym samym wymagane zależności.



Rysunek 1.1. Klient używa usługi reprezentowanej przez interfejs

Prawdopodobnie już zdarzało Ci się stosować taki sposób kodowania. Jednak dopiero teraz, kiedy wykorzystamy framework, który w spójny sposób używa tego modelu programowego w obrębie całej aplikacji, przekonamy się, jak duże możliwości on zapewnia. Kod składający się z luźno powiązanych komponentów jest znacznie prostszy do testowania, utrzymania i zrozumienia.

Jak to zrobić?

Aby stosować ten model projektowy, wcale nie trzeba korzystać z lekkich kontenerów. W celu rozdzielenia elementów programu przy wykorzystaniu wstrzykiwania zależności należy wykonać trzy czynności:

1. Stworzyć interfejs, który będzie reprezentować usługę.
2. Dodać do klienta właściwość odwołującą się do usługi.
3. Przy wykorzystaniu dodatkowego frameworka lub własnego kodu stworzyć usługę i przypisać odpowiednią wartość właściwości klienta.

Pierwszym krokiem jest wydzielenie interfejsu. Teraz zmienimy nazwę pliku, definicji klasy oraz konstruktora `RentABike` na `ArrayListRentABike` (listing 1.4) i stworzymy interfejs (listing 1.5).

Listing 1.4. ArrayListRentABike.java (wcześniej RentABike.java)

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListRentABike implements RentABike {
   private String storeName;
   final List bikes = new ArrayList();

   public ArrayListRentABike(String storeName) {
      this.storeName = storeName;
      bikes.add(new Bike("Shimano", "Roadmaster", 20, "11111", 15,
```

```
                "Dobry"));
        bikes.add(new Bike("Cannondale", "F2000 XTR", 18, "22222",12,
                "Doskonały"));
        bikes.add(new Bike("Trek","6000", 19, "33333", 12.4,
                "Dobry"));
    }

    public ArrayListRentABike() {
        this("Rowery Bruce'a");
    }

    public void setStoreName(String storeName) { this.storeName =
    storeName; }
    public String getStoreName() { return this.storeName; }

    public String toString() { return "RentABike: " + storeName; }

    public List getBikes() { return bikes; }

    public Bike getBike(String serialNo) {
        Iterator iter = bikes.iterator();
        while(iter.hasNext()) {
            Bike bike = (Bike)iter.next();
            if(serialNo.equals(bike.getSerialNo())) return bike;
        }
        return null;
    }
}
```

Listing 1.5. RentABike.java

```
import java.util.*;

public interface RentABike {
    List getBikes();
    Bike getBike(String serialNo);
}
```

Kolejnym elementem programu jest widok, przedstawiony na listingu 1.6. Warto zauważyć, że wydzieliliśmy metody, które wyświetlają informacje o rowerze w wierszu poleceń. Dodaliśmy także do widoku właściwość, w której można zapisać obiekt typu RentABike.

Teraz będziemy mogli zobaczyć usługę RentABike udostępnioną w formie właściwości. Nieco później jej wartość zostanie określona przez framework Spring.

Listing 1.6. CommandLineView.java

```
import java.util.*;

public class CommandLineView {
    private RentABike rentaBike;
    public CommandLineView() {}

    public void setRentaBike(RentABike rentaBike) {
```

```
            this.rentaBike = rentaBike;
        }

        public RentABike getRentaBike() { return this.rentaBike; }

        public void printAllBikes() {
            System.out.println(rentaBike.toString());
            Iterator iter = rentaBike.getBikes().iterator();
            while(iter.hasNext()) {
                Bike bike = (Bike)iter.next();
                System.out.println(bike.toString());
            }
        }
    }
```

Tutaj możemy zaobserwować działanie wstrzykiwania zależności. Dodatkowy kod, w tym przypadku napisany przez nas, tworzy oba obiekty i wstrzykuje referencję typu RentABike do obiektu widoku.

Ostatnim elementem aplikacji jest asembler, który tworzy wszystkie obiekty i określa wartość właściwości (listing 1.7).

Listing 1.7. RentABikeAssembler.java

```
public class RentABikeAssembler {
    public static final void main(String[] args) {
        CommandLineView clv = new CommandLineView();
        RentABike rentaBike = new ArrayListRentABike("Rowery Bruce'a");
        clv.setRentABike(rentaBike);
        clv.printAllBikes();
    }
}
```

Teraz można skompilować aplikację, wykonując następujące polecenie:

```
C:\PozyczRowerApp\src> javac -d ..\out *.java
```

Katalog wynikowy będzie zawierać następujące, skompilowane pliki klas:

Katalog: C:\PozyczRowerApp\out

```
2005-12-31  15:28    <DIR>          .
2005-12-31  15:28    <DIR>          ..
2006-01-02  15:02             1 737 Bike.class
2006-01-02  15:02             1 477 ArrayListRentABike.class
2006-01-02  15:02               186 RentABike.class
2006-01-02  14:58               944 CommandLineView.class
2006-01-02  14:58               496 RentABikeAssembler.class
```

Aplikację można uruchomić w następujący sposób:

```
C:\PozyczRowerApp\out>java RentABikeAssembler
RentABike: Rowery Bruce'a
Rower : producent -- Shimano
: model -- Roadmaster
: rama -- 20
: numer seryjny -- 11111
```

```
: waga -- 15.0
: status -- Dobry.

Rower : producent -- Cannondale
: model -- F2000 XTR
: rama -- 18
: numer seryjny -- 22222
: waga -- 12.0
: status -- Doskonały.

Rower : producent -- Trek
: model -- 6000
: rama -- 19
: numer seryjny -- 33333
: waga -- 12.4
: status -- Dobry.
```

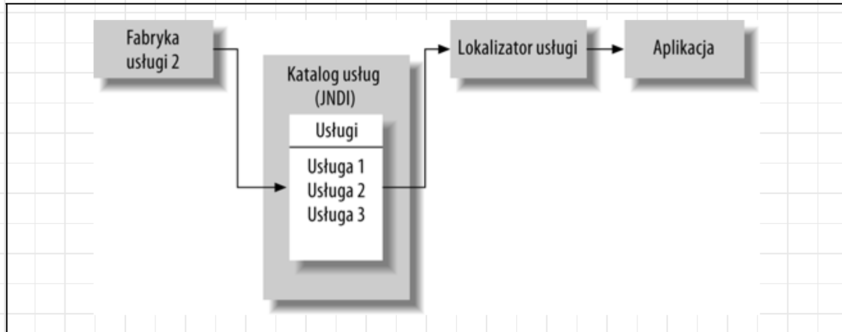
Jak to działa?

Właśnie zobaczyłeś przykład wstrzykiwania zależności poza środowiskiem lekkiego kontenera. Choć środowiska twórców i użytkowników lekkich kontenerów robią wiele hałasu w związku z wstrzykiwaniem zależności, to jednak pomysły leżące u podstaw tego rozwiązania są całkiem proste. Pisząc programy, należy używać interfejsów, a zależności nie określać samemu, lecz przy wykorzystaniu dodatkowego narzędzia.

W naszych przykładach, asemblery zastąpimy w końcu frameworkiem Spring. Kiedy to się stanie, framework będzie samodzielnie wywoływać konstruktory używanych obiektów i określać występujące pomiędzy nimi zależności. Niemniej jednak na razie musimy poświęcić tej wersji aplikacji nieco więcej uwagi.

A co z...

... lokalizatorami usług lub obiektami fabrykującymi? Oczywiście wstrzykiwanie zależności nie jest jedynym sposobem rozwiązywania zależności występujących pomiędzy obiektami. W rzeczywistości nie jest to jedyny dobry sposób rozwiązywania tego problemu. Osoby korzystające z technologii J2EE wykorzystują zazwyczaj lokalizatory usług (ang. *service locators*). Używając tego wzorca (przedstawionego na rysunku 1.2), można przedstawić zależność w formie interfejsu, zarejestrować ją w słowniku, a następnie odszukać przy wykorzystaniu dodatkowej klasy pomocniczej nazywanej lokalizatorem (ang. *locator*). Nie jest to wcale złe rozwiązanie.



Rysunek 1.2. Aplikacje J2EE zarządzają zależnościami przy wykorzystaniu lokalizatorów usług

Strategia wstrzykiwania zależności pozwala na zastosowanie spójnego rozwiązania i całkowite oddzielenie zależności od tworzonej aplikacji. W dalszej części rozdziału przekonasz się, w jaki sposób rozwiązanie to może pomóc w testowaniu aplikacji i tworzeniu programów zapewniających dużą łatwość utrzymania i konfiguracji.

W rzeczywistości wcześniejsze lekkie kontenery, takie jak Avalon, wykorzystywały właśnie to rozwiązanie. Większość nowoczesnych kontenerów udostępnia mechanizmy służące do wyszukiwania zależności, choć zazwyczaj preferowane są inne sposoby ich rozwiązywania.

Automatyzacja przykładu

Najwyższy czas na małe porządki. Aby móc pójść dalej, konieczne jest zastosowanie automatyzacji. Prawdopodobnie używasz już programu Ant. Stanowi on standaryzowany sposób organizacji wszystkich zadań, jakie należy wykonać w celu skompilowania i przygotowania aplikacji. Jeśli jeszcze nie używasz tego programu, to będziesz musiał zacząć.

Ant stał się wszechobecny. Aby móc posługiwać się narzędziami programistycznymi języka Java, konieczne będzie postanie używanego przez nie „języka”. Nie mamy zamiaru zamieszczać w tym rozdziale uzasadnienia, dlaczego warto korzystać z programu Ant, gdyż zapewnie i tak już wiele na ten temat czytałeś.

Jak to zrobić?

Nie ma potrzeby pobierania pakietu instalacyjnego programu Ant — można wykorzystać ten, który jest dostępny wraz z frameworkiem Spring (*http://springframework.org/*). W celu wykonania wszystkich przykładów zamieszczonych w tej książce będziesz potrzebował frameworka Spring w wersji 1.1 lub późniejszej. Podczas jego instalacji należy postępować zgodnie z zaleceniami dotyczącymi używanego systemu operacyjnego.

Kolejnym krokiem po zainstalowaniu frameworka Spring będzie zorganizowanie takiej struktury katalogów, która będzie wygodna zarówno dla frameworka Spring, jak i dla tworzonej aplikacji. Zalecamy zastosowanie takiej samej organizacji, jaka została użyta w przykładowej aplikacji dostarczanej wraz z frameworkiem Spring, dzięki czemu zyskamy możliwość przeanalizowania gotowego, działającego przykładu. Poniżej przedstawiliśmy listę katalogów, z których będziemy korzystać:

src

W tym katalogu przechowywane są wszystkie kody źródłowe aplikacji. Jak zwykle struktura podkatalogów umieszczonych wewnątrz tego katalogu powinna odpowiadać strukturze pakietów.

test

W tym katalogu przechowywane są wszystkie testy modułów. Więcej informacji dotyczących JUnit można znaleźć w ostatnim ćwiczeniu zamieszczonym pod koniec tego rozdziału.

db

W tym katalogu umieszczane są wszelkie zasoby związane z bazami danych — skrypty, konfiguracje oraz kody. Niektóre z nich będą plikami konfiguracyjnymi, inne plikami pomagającymi stworzyć bazę danych o odpowiedniej strukturze, a jeszcze inne pozwolą na zapisanie w bazie informacji testowych. Jeśli nasza aplikacja ma współpracować z kilkoma różnymi bazami danych, to dla każdej z tych baz w katalogu *db* zostanie utworzony odrębny podkatalog.

war

Plik *war* jest typową jednostką wdrażania aplikacji sieciowych. Jeśli używasz kontenera J2EE lub kontenera serwletów, takiego jak Tomcat, to w tym katalogu będzie umieszczany plik konfiguracyjny *web.xml*. W tym katalogu będą umieszczane także pliki konfiguracyjne używane przez framework Spring. Informacje o plikach konfiguracyjnych Springa zostały podane w ćwiczeniach zamieszczonych w dalszej części książki.

Na początek rozmieść pliki wchodzące w skład naszej aplikacji w następujący sposób:

- Plik *RentABike.java* — w katalogu *src*, w podkatalogach odpowiadających strukturze pakietów.
- Plik *ArrayListRentABike.java* — w katalogu *src* wraz z plikiem *RentABike.java*.
- Plik *Bike.java* — w katalogu *src* wraz z plikiem *RentABike.java*.
- Plik *CommandLineView.java* — w katalogu *src* wraz z plikiem *RentABike.java*.

W końcu będzie Ci potrzebny skrypt programu Ant, który należy umieścić w głównym katalogu projektu. Listing 1.8 przedstawia skrypt od którego zaczniemy:

To właśnie w tym pliku przekażesz programowi Ant informacje dotyczące położenia plików klas wchodzących w skład aplikacji. Można je dołączyć do pliku war aplikacji lub wskazać bezpośrednio ich wersje instalacyjne i poinformować użytkowników, jakie pliki muszą wdrożyć.

Listing 1.8. build.xml

```
<?xml version="1.0"?>
<project name="RentABike" default="compile" basedir=".">

   <property name="src.dir" value="src"/>
   <property name="test.dir" value="test"/>
   <property name="war.dir" value="war"/>
   <property name="class.dir" value="${war.dir}/WEB-INF/classes"/>

   <target name="init">
      <mkdir dir="${class.dir}"/>
   </target>

   <target name="compile" depends="init"
      description="Compiles all source code.">
      <javac srcdir="${src.dir}" destdir="${class.dir}"/>
   </target>
```

```
    <target name="clean" description="Erases contents of classes dir">
      <delete dir="${class.dir}"/>
    </target>

</project>
```

W celu wykonania powyższego skryptu programu Ant należy przejść do katalogu o nazwie *C:\PozyczRowerApp* i wydać następujące polecenie:

```
C:\PozyczRowerApp>ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\PozyczRowerApp\war\WEB-INF\classes

compile:
    [javac] Compiling 5 source files to C:\PozyczRowerApp\war\
    WEB-INF\classes

BUILD SUCCESSFUL
Total time: 2 seconds
```

Właśnie to nazywam automatyzacją. Ograniczyliśmy całkowitą liczbę klawiszy, jakie należy nacisnąć w celu skompilowania i przygotowania aplikacji — w tym momencie cały proces wymaga naciśnięcia dokładnie czterech klawiszy.

Jak to działa?

Program Ant przygotował aplikację w jednym, zautomatyzowanym kroku. Jak na razie nie jest to co prawda wielkim ułatwieniem, jednak znaczenie tej automatyzacji będzie stopniowo rosło, wraz ze wzrostem stopnia złożoności poszczególnych czynności wykonywanych w ramach kompilacji i przygotowywania aplikacji. Później będziesz chciał, by system samoczynnie wykonywał testy modułów, dodawał operacje związane z prekompilacją, takie jak wzbogacenie kodów bajtowych przez JDO (patrz rozdział 5.) lub kopiował pliki konfiguracyjne w odpowiednie miejsca. Można także wykonywać zadania specjalne w celu zainicjowania bazy danych lub wdrożenia pliku *war* na serwer aplikacji.

Szczerze mówiąc, budujemy tę aplikację w zintegrowanym środowisku programistycznym o nazwie IDEA, stworzonym przez firmę JetBrains. Według nas dysponuje ono najlepszymi dostępnymi możliwościami refaktoringu. Ale się nie przejmuj — każdą prezentowaną wersję aplikacji testowaliśmy także przy wykorzystaniu skryptów programu Ant.

A co z…

… faktem, że niektóre zintegrowane środowiska programistyczne w ogóle nie muszą korzystać z programu Ant? Jeśli chcesz używać takich środowisk programistycznych i nie przejmować się programem Ant, to będziesz musiał upewnić się, że:

- Kod będzie rozmieszczony w katalogach o takiej samej strukturze jak zastosowana przez nas, by nie pojawiły się żadne problemy z pakietami.

- Zintegrowane środowisko programistyczne będzie miało dostęp do wszystkich plików *.jar*, z których będziemy korzystać.
- W dalszych rozdziałach będziesz musiał znaleźć sposób wdrażania i uruchamiania aplikacji sieciowej, oraz wykonywania testów JUnit.

Od tej chwili aż do końca książki będziemy informować, kiedy konieczne będzie dodanie nowego katalogu, wykonanie aplikacji lub udostępnienie nowej biblioteki. Nie będziemy jednak opisywać, jak należy to zrobić, zakładając, że będziesz w stanie wykorzystać do tego celu możliwości używanego środowiska programistycznego lub będziesz wiedzieć, jak to uczynić bezpośrednio w skrypcie programu Ant.

Wstrzykiwanie zależności przy wykorzystaniu frameworka Spring

Już niemal przygotowaliśmy naszą aplikację do zastosowania frameworka Spring. Czas go pobrać i użyć. W tym ćwiczeniu zastąpimy obiekt `RentABikeAssembler` frameworkiem Spring.

Kiedy zacząłem używać Springa zamiast J2EE, moje życie się odmieniło. Stałem się znacznie bardziej produktywny. Okazało się, że pisanie kodu łatwiej stało się dla mnie łatwiejsze, a tworzony przeze mnie kod jest łatwiejszy do zrozumienia dla moich klientów. Z kolei uproszczenie kodu niejednokrotnie pozwalało na jego szybsze działanie. Poza tym wprowadzanie jakichkolwiek zmian w kodzie stało się nieporównanie łatwiejsze. W rzeczywistości napisałem książkę o znaczeniu i zaletach lekkich frameworków takich jak Spring; nosi ona nazwę *Better, Faster, Lighter Java* i została wydana przez wydawnictwo O'Reilly.

Jak to zrobić?

W pierwszej kolejności należy pobrać pakiet instalacyjny frameworka Spring. Można go znaleźć na witrynie *http://springframework.org/*. Stamtąd należy przejść na witrynę *Sourceforge*, na której można znaleźć wersję frameworka dostosowaną do używanego systemu operacyjnego (podczas pisania niniejszej książki używaliśmy wersji Spring 1.1). Następnie trzeba

dodać do projektu nowy katalog — *war\WEB-INF\lib* — i umieścić w nim biblioteki frameworka Spring (całą zawartość katalogu *dist* dostępnego w pakiecie instalacyjnym frameworka Spring).

Modyfikacja poprawnie zaprojektowanej aplikacji bazującej na zwyczajnych, starych obiektach Javy — w skrócie POJO (ang. *Plain Old Java Objects*) — i przystosowanie jej do korzystania z frameworka Spring nie przysparza najmniejszych problemów. Wszystko sprowadza się do wykonania poniższych czynności:

- Zmodyfikowania kodu tak, aby korzystał ze wstrzykiwania zależności. Obiekty modelu są komponentami, a usługi — aspektami. Zazwyczaj jednak będziesz używał jedynie komponentów.
- Usunięcia kodu, który tworzy obiekty i rozwiązuje zależności.
- Napisania pliku konfiguracyjnego opisującego używane komponenty i aspekty.
- Odwołania się do napisanego kodu za pośrednictwem frameworka Spring.

Ponieważ poszczególne elementy naszej aplikacji są już przygotowane do wykorzystania wstrzykiwania zależności, zatem zastosowanie frameworka Spring będzie łatwym ćwiczeniem. Po prostu zastąpimy nasz asembler wersją korzystającą z frameworka i stworzymy plik konfiguracyjny, który zostanie umieszczony w katalogu *\war\WEB-INF*.

Plik konfiguracyjny został przedstawiony na listingu 1.9.

Listing 1.9. RentABike-context.xml

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

<beans>
   <bean id="rentaBike" class="com.springbook.ArrayListRentABike">
      <property name="storeName"><value>Rowery Bruce'a</value></property>
   </bean>

   <bean id="commandLineView" class="com.springbook.CommandLineView">
      <property name="rentaBike"><ref bean="rentaBike"/></property>
   </bean>

</beans>
```

Aby uruchomić skrypt przygotowujący aplikację, należy przejść do katalogu *PozyczRowerApp* i wydać następujące polecenie:

```
C:\PozyczRowerApp\ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\PozyczRowerApp\war\WEB-INF\classes

compile:
    [javac] Compiling 5 source files to C:\PozyczRowerApp\war\WEB-
INF\classes

BUILD SUCCESSFUL
Total time: 2 seconds
```

Na listingu 1.10 został przedstawiony nowy asembler, który zastąpił wcześniejszą klasę `RentABikeAssembler`.

Listing 1.10. RentABikeAssembler.java

```
package com.springbook;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class RentABikeAssembler {
   public static final void main(String[] args) {
      ClassPathXmlApplicationContext ctx = new
      ClassPathXmlApplicationContext("RentABike-context.xml");
      CommandLineView clv =
      (CommandLineView)ctx.getBean("commandLineView");
      clv.printAllBikes();
   }
}
```

Jak to działa?

Być może drapiesz się po głowie i zastanawiasz, co w tym wszystkim jest takiego niezwykłego. Jednak te niewielkie zmiany architektury będą miały znaczący wpływ na cykl życia aplikacji. Korzyści wynikające z ich wprowadzenia można zauważyć niemal od razu. Nie będę Cię zanudzać i opisywać ich dokładnie. Zamiast tego napiszę o tym, co się dzieje w ukryciu — w sposób niezauważalny dla programisty.

A co z...

... Pico, Hive Mind oraz Avalonem? To wszystko są lekkie kontenery. Każdy z nich ma swoje silne i słabe punkty. Ani Avalon, ani Hive Mind nie uzyskały na tyle dużej masy krytycznej, abyś chciał myśleć o ich zastosowaniu, zwłaszcza jeśli chcesz używać usług, które są w stanie współdziałać ze sobą. Obecnie największe udziały w rynku mają frameworki Spring oraz Pico. Programiści poszukujący samodzielnego kontenera zazwyczaj decydują się na wykorzystanie Pico, jednak Spring posiada najbardziej rozbudowane i kompletne usługi dodatkowe, takie jak deklaratywne transakcje i bogate strategie zapewniania trwałości.

Tworzenie testu

Każdy z rozdziałów niniejszej książki zostanie zakończony stworzeniem testu. Prawdę mówiąc, jeśli jesteś programistą, który głęboko wierzy w pisanie programów w oparciu o testy, to testy modułów powinieneś tworzyć *w pierwszej kolejności*. Wielu z Was kupiło tę książkę, gdyż Spring jest w stanie poprawić łatwość testowania aplikacji. W zasadzie poprawa możliwości testowania od samego początku stanowiła jedno z podstawowych założeń architektury frameworka Spring.

Automatyzacja testów da Ci większą pewność co do tego, że Twój kod działa poprawnie, a co więcej, że będzie działać poprawnie także po wprowadzeniu zmian. W rzeczywistości wszyscy Twoi szefowie czytają te same książki, które informują, że utrzymywanie dużych zespołów testujących jest kosztowne. My musimy przejąć tę pałeczkę. Nie ma żadnego efektywnego sposobu testowania, który by nie korzystał z automatyzacji. Niemniej jednak istnieje pewnie poważny problem. Wiele spośród obecnie stosowanych architektur, takich jak EJB oraz Struts, raczej nie ułatwiają testowania. Ich czas ładowania jest długi, a tworzenie obiektów zastępczych (ang. *mock objects*) jest trudne.

Jednak Spring diametralnie zmienia tę sytuację. Każdy obiekt może być wykorzystany poza środowiskiem kontenera. Co więcej, ze względy na to, że kontener jest taki „lekki", koszty związane z jego uruchamianiem

są pomijalnie małe. To niezwykle duża zaleta, zwłaszcza jeśli musimy testować tworzony kod. W końcu Spring zachęca i wspiera model programowania o bardzo słabych powiązaniach pomiędzy komponentami.

Jak to zrobić?

Test modułu (test jednostkowy, ang. *unit test*) można sobie wyobrazić jako kolejnego klienta aplikacji. Testy przyjmują pewne założenia odnośnie warunków, jakie powinny być spełnione w przypadku, gdy aplikacja będzie działać poprawnie. Na przykład, jeśli dodajesz obiekt do listy, to jej wielkość powinna się zwiększyć o jeden. Następnie test można wykonać samodzielnie podczas dodawania do kodu kolejnej funkcji, bądź też, podczas testowania starego kodu. Testy można także wykonywać jako jeden z elementów całego procesu kompilacji i przygotowywania aplikacji. W takim przypadku, jeśli założenia przyjęte w testach nie zostaną spełnione, to cały proces przygotowywania aplikacji zakończy się niepowodzeniem.

Każdy test modułu jest klasą dziedziczącą po klasie `TestCase`. Listing 1.11 przedstawia test stanowiący klienta fasady.

Listing 1.11. RentABikeTest.java

```
import java.util.List;
import junit.framework.TestCase;

public class RentABikeTest extends TestCase {
   private RentABike rentaBike;

   public void setUp() {
      rentaBike = new ArrayListRentABike("Rowery Bruce'a");
   }

   public void testGetName() {
      assertEquals("Rowery Bruce'a", rentaBike.getStoreName());
   }

   public void testGetBike() {
      Bike bike = rentaBike.getBike("11111");
      assertNotNull(bike);
      assertEquals("Shimano", bike.getManufacturer());
   }

   public void testGetBikes() {
      List bikes = rentaBike.getBikes();
```

```
        assertNotNull(bikes);
        assertEquals(3, bikes.size());
    }
}
```

Następnie, aby uruchomić test, konieczne będzie zmodyfikowanie skryptu programu Ant. Listing 1.12 przedstawia zmienione zadania inicjalizacji i kompilacji oraz dodatkowe zadanie odpowiadające za kompilację testu (trzeba zauważyć, iż wyrażenia *ŚCIEŻKA_DO_SPRING* i *ŚCIEŻKA_DO_JUNIT* należy zastąpić faktycznymi ścieżkami).

Listing 1.12. build.xml

```
<property name="test.class.dir" value="${test.dir}/classes"/>
<property name="spring.dir" value="ŚCIEŻKA_DO_SPRING"/>

<path id="bikestore.class.path">
  <fileset dir="${spring.dir}/dist">
    <include name="*.jar" />
  </fileset>

  <pathelement location="${spring.dir}/lib/jakarta-commons/commons-
   logging.jar"/>
  <pathelement location="${spring.dir}/lib/log4j/log4j-1.2.8.jar"/>
  <pathelement location="${spring.dir}/lib/j2ee/servlet.jar"/>[1]
  <dirset dir="${basedir}"/>
  <dirset dir="${class.dir}"/>
</path>

<path id="run.class.path">
   <path refid="bikestore.class.path"/>
   <dirset dir="${class.dir}"/>
</path>

<path id="junit.class.path">
   <path refid="run.class.path"/>
   <pathelement location="ŚCIEŻKA_DO_JUNIT"/>
</path>

<target name="init">
   <mkdir dir="${class.dir}" />
   <mkdir dir="${test.class.dir}" />
</target>

<target name="compile" depends="init"
```

[1] W najnowszej wersji frameworka Spring dostępnej w chwili przygotowywania polskiego wydania niniejszej książki (1.2.6) zamiast `servlet.jar` powinno być `servlet-api.jar` — *przyp. red.*

```
        description="Compiles all source code.">
        <javac srcdir="${src.dir}"
             destdir="${test.class.dir}"
               classpathref="bikestore.class.path"/>
</target>

<target name="compile.test" depends="init" description="Compiles all unit
test source">
   <javac srcdir="${test.dir}"
          destdir="${test.class.dir}"
          classpathref="junit.class.path"/>
</target>
```

Listing 1.13 przedstawia zadanie odpowiadające za wykonanie testu.

```
<target name="test" depends="compile, compile.test"
   description="Runs the unit tests">
   <junit printsummary="withOutAndErr" haltonfailure="no"
      haltonerror="no" fork="yes">
      <classpath refid="junit.class.path"/>
      <formatter type="xml" usefile="true" />
      <batchtest todir="${test.dir}">
         <fileset dir="${test.class.dir}">
            <include name="*Test.*"/>
         </fileset>
      </batchtest>
   </junit>
</target>
```

A oto wyniki testu:

```
Buildfile: build.xml

init:

compile:

compile.test:

test:
    [junit] Running RentABikeTest
    [junit] Test run: 3, Failures: 0, Errors: 0; Time elapsed: 0.36 sec

BUILD SUCCESSFUL
Total time: 2 seconds
```

Jak to działa?

Ant właśnie przygotował całą aplikację, w tym także test. Następnie testy zostały wykonane. Wszystko zakończyło się powodzeniem. Gdyby testów nie udało się przeprowadzić pomyślnie, to aby cały proces przygotowywania aplikacji mógł się poprawnie zakończyć, konieczne byłoby

takie poprawienie jej kodu, by warunki określane w testach zostały spełnione. W ten sposób można wychwytywać i poprawiać niewielkie błędy, zanim przerodzą się one w duże problemy.

W następnym rozdziale zaczniemy na poważnie korzystać z frameworka Spring. Stworzymy prawdziwy interfejs użytkownika dla naszej aplikacji, wykorzystując do tego celu szkielet Web MVC wchodzący w skład Springa. Pokażemy także, w jaki sposób można zintegrować ze Springiem istniejące komponenty interfejsu użytkownika.