Eberhard Wolff

SpringSource dm Server

Der SpringSource dm Server ist der erste OSGi-basierte Open-Source-Applikationsserver.

Dieses Dokument gibt eine umfangreiche Einführung in seinen Aufbau und seinen Einsatz. Dabei wird zunächst auf die grundlegenden Technologien wie Spring und OSGi eingegangen, um dann die Features des dm Server näher zu erläutern. Anhand von zwei Beispielen wird gezeigt, wie man mit dem dm Server Anwendungen entwickeln kann und welche Auswirkungen das auf die Architektur der Anwendungen hat. Zum Abschluss wird auch auf die Installation und den Betrieb des Servers eingegangen.

dpunkt.verlag

Leser

· **Java-Entwickler**

ISBN 978-3-89864-955-1

www.dpunkt.de

Inhalt

Was sind dpunkt.ebooks?

Die dpunkt.ebooks sind Publikationen im PDF-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken. Sie benötigen zum Ansehen den Acrobat Reader (oder ein anderes adäquates Programm).

dpunkt.ebooks können Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt. büchern her kennen.

Was darf ich mit dem dpunkt.ebook tun?

Die Datei ist nicht kopiergeschützt, kann also für den eigenen Bedarf beliebig kopiert werden. Es ist jedoch nicht gestattet, die Datei weiterzugeben oder für andere zugänglich in Netzwerke zu stellen.

Sie erwerben also eine Ein-Personen-Nutzungslizenz.

Wenn Sie mehrere Exemplare des gleichen E-Books kaufen, erwerben Sie damit die Lizenz für die entsprechende Anzahl von Nutzern.

Um Missbrauch zu reduzieren, haben wir die PDF-Datei mit einer PIN und einem Wasserzeichen (Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer) versehen.

Bitte beachten Sie, dass die Inhalte der Datei in jedem Fall dem Copyright des Verlages unterliegen.

Wie kann ich dpunkt.ebooks kaufen und bezahlen?

Legen Sie die E-Books in den Warenkorb. (Aus technischen Gründen, können im Warenkorb nur gedruckte Bücher ODER E-Books enthalten sein.)

Downloads und E-Books können sie bei dpunkt per Paypal bezahlen. Wenn Sie noch kein Paypal-Konto haben, können Sie dieses in Minutenschnelle einrichten (den entsprechenden Link erhalten Sie während des Bezahlvorgangs) und so über Ihre Kreditkarte oder per Überweisung bezahlen.

Wie erhalte ich das dpunkt.ebook?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene E-Mail-Adresse eine Bestätigung von Paypal sowie eine E-Mail vom dpunkt.verlag mit dem folgenden Inhalt:

- Downloadlinks für die gekauften Dokumente
- PINs für die gekauften Dokumente
- eine PDF-Rechnung für die Bestellung

Die Downloadlinks sind zwei Wochen lang gültig. Die Dokumente selbst sind durch eine PIN geschützt und mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag:

hallo@dpunkt.de

Einleitung

Warum dm Server?

Enterprise Java, und insbesondere Java EE, ist eine etablierte Basis für die Implementierung von Geschäftsanwendungen. Das Programmiermodell, hier vor allem EJB, ist allerdings schon vor längerer Zeit in die Kritik geraten. Das Ergebnis war eine zunehmende Rückbesinnung auf die Nutzung einfacher Java-Objekte und die Nutzung von Technologien wie Spring [Spring], um Systeme aus solchen einfachen Java-Objekten aufzubauen. Dabei bleibt aber das Deployment-Modell unangetastet. Die Anwendungen werden meistens als WAR-Dateien oder EAR-Dateien auf einem Java-EE-Application-Server betrieben, die sich ja auch mittlerweile in den 10 Jahren des Bestehens dieses Standards auf breiter Front etabliert haben.

Java-EE-Deployment: Probleme

Dadurch bleiben jedoch eine Probleme ungelöst:

- Die Java-EE-Lösung ist mächtig, aber daher auch komplex. In den meisten Fällen reicht eine wesentlich einfachere Umgebung. Daher haben sich einfache Java-Webserver wie Tomcat mittlerweile als beliebteste Plattform für Enterprise Java etabliert, obwohl sie keine vollständige Java-EE-Implementierung sind, sondern sich auf Webanwendungen begrenzen.
- Selbst die kleinste Änderung an einer Anwendung führt dazu, dass man die gesamte Deployment-Einheit (z. B. das WAR) neu ausliefern muss. Das kann bedeuten, dass eine kleine Änderung an einer Webseite zu einer neuen Auslieferung der gesamten Anwendung inklusive aller Webseiten und der kompletten Logik führt, obwohl das meiste überhaupt nicht angetastet worden ist. Für diese neue Auslieferung muss man die gesamte Anwendung neu compilieren und erstellen. Um mögliche Fehler beim Erstellen der Anwendung auszuschließen, muss man sie zunächst testen. Und das Deployment selbst führt zu einem temporären Ausfall der Anwendung, da sie ja für das Deployment neu gestartet werden muss. Man kann einige dieser Probleme mit geschickten Deployment-Modellen lösen, die zum Beispiel zunächst die neue und die alte Version parallel betreiben. Dennoch wären feingranularere Deployment-Einheiten wünschenswert, so dass man kleinere Einheiten neu deployen kann.
- Gleichzeitig ist es so, dass man Logik oft über verschiedene Deployment-Einheiten gemeinsam nutzen will. Beispielsweise kann es gewünscht sein, dass man die Logik einer Anwendung über eine Weboberfläche und über einen Web Service nutzen kann. Das ist aber nicht einfach, weil Deployment-Einheiten komplett voneinander isoliert sind. Man kann also als Lösung den Web Service und die Weboberfläche zusammen mit der Logik in eine Deployment-Einheit zusammenfassen. Beispielsweise kann man sie beide in ein WAR zusammenfassen oder in ein EAR. Dann hat man aber eine größere Deployment-Einheit. Das führt dazu, dass das Deployment länger dauert. Außerdem führt eine Änderung an der Weboberfläche nun auch ein Redeployment des Web Service. Dieser muss also nun auch bei einem Redeployment getestet werden und steht während des Deployments nicht zur Verfügung. Eine andere Möglichkeit ist es, die Weboberfläche den Web Service nutzen zu lassen. Dann hat man zwei Deployment-Einheiten, aber auch einen großen Overhead. Während nämlich zuvor einfach der Service durch einen lokalen Methodenaufruf angesprochen werden konnte, muss

man nun durch die Netzwerkverbindung gehen und dafür die Parameter bzw. Ergebnisse entsprechend serialisieren. Also ist dies auch keine gute Lösung. Hier fehlt die Möglichkeit, die Deployment-Einheiten miteinander kommunizieren zu lassen.

- Außerdem hat Java keine Versionierung: Man kann nicht mehrere Versionen einer Bibliothek gleichzeitig nutzen und man kann auch nicht definieren, dass man bestimmte Versionen bestimmter Bibliotheken benötigt. Das führt zu Problemen und im Extremfall sogar dazu, dass die Anwendung überhaupt nicht laufen kann, weil man eine Bibliothek in einer bestimmten Version (z. B. 1.0) nutzen will, aber eine andere Bibliothek, die man gleichzeitig nutzen will, eine andere Version (z. B. 1.1) dieser Bibliothek erfordert.

Letztendlich besteht das Problem darin, dass Anwendungen heute zwar meistens modular entwickelt werden, aber im Deployment wird dann alles wieder in eine große Datei eingepackt und es entsteht daraus ein Monolith. Anders gesagt: Man macht sich zuvor viele Gedanken über Architektur und die Aufteilung in Schichten, aber zur Laufzeit und im Betrieb ist davon dann nichts mehr übrig. Dieses Problem zu lösen ist nicht ganz einfach. Wenn man in der Lage sein will, Teile einer Anwendung neu zu deployen, ohne alle abhängigen Elemente neu zu deployen, dann muss man damit umgehen können, dass bestimmte Teile der Anwendung zur Laufzeit plötzlich nicht mehr zur Verfügung stehen, weil sie gerade neu deployt werden. Dabei geht es um Code. Man muss also zur Laufzeit bestimmte Codeanteile austauschen können.

Die Lösung: OSGi

Die OSGi Service Platform [OSGi] schickt sich an, dieses Problem zu lösen. Sie bietet die Möglichkeit, den Code in Bundles aufzuteilen. Diese sind versioniert und können zur Laufzeit installiert und deinstalliert werden. Prinzipiell sind damit die oben genannten Probleme gelöst: Man hat feingranulare Deployment-Einheiten, die auch über lokale Methodenaufrufe miteinander kommunizieren können. Außerdem ist OSGi heute schon in breiter Nutzung: Die Eclipse-Entwicklungsumgebung basiert auf OSGi, und im Embedded-Bereich gibt es sowieso schon zahlreiche Nutzungsbeispiele wie beispielsweise für die Steuerung von Fahrzeugen oder Software für PDAs und Mobiltelefone. Also liegt es auf der Hand, dieses Technologie auch im Enterprise-Java-Bereich zu nutzen.

OSGi & Enterprise-Java: Probleme

Aber ganz so einfach ist es leider nicht: Die Nutzung von OSGi führt schon beispielsweise bei der Verwendung von typischen Enterprise-Java-Bibliotheken wie Hibernate zu Problemen. Außerdem wäre es wünschenswert, wenn man eine Umgebung hätte, in der man die Anwendungen deployen kann. Typische Applikationsserver bieten aber keine Unterstützung für OSGi – auch wenn praktisch alle OSGi intern nutzen. Genau diese Lücke füllt der dm Server: Er bietet eine Ablaufumgebung für OSGi-Anwendungen und löst dabei auch die Probleme, die man bei der Nutzung von OSGi für Enterprise-Java-Anwendung sonst hat.

Über dieses Dokument

Das Ziel dieses Dokuments ist es, eine Einführung in den dm Server zu geben. Dabei werden die bereits angeschnittenen Probleme und Lösungen detailliert dargestellt.

Zunächst wird in Kapitel 1 ein Beispiel eingeführt, das in den folgenden Abschnitten in Form von Beispielcode immer wieder auftaucht. Kapitel 2 gibt eine kurze Einführung in das Spring-Framework, das die Basis des Programmiermodells des dm Servers darstellt. Kapitel 3 führt OSGi ein, auf dem das Deployment-Modell des dm Servers basiert. Eine einfache Integration zwischen OSGi und Spring stellt das Projekt *Spring Dynamic Module for the OSGi Platform* dar, das in Kapitel 4 erläutert wird. Kapitel 5

führt dann schließlich in den dm Server selbst ein. Kapitel 6 zeigt anhand der Beispielanwendung, wie man eine Anwendung praktisch mit dem dm Server umsetzen kann. Kapitel 7 zeigt ein weiteres Beispiel: die Petclinic, die man direkt mit dem Server zusammen herunterladen kann. Schließlich geht Kapitel 8 auf die Installation und den Betrieb des dm Servers ein.

1 Die Beispielanwendung

Um die in diesem Dokument erläuterten Konzepte auch gleich in einer praxisnahen Verwendung zu sehen, wird im gesamten Dokument ein durchgängiges Beispiel verwendet. Da die meisten individuellen Softwareprojekte im Bereich der Geschäftsanwendungen liegen, steht eine solche Anwendung auch hier im Mittelpunkt. Man kann natürlich nicht erwarten, dass dieses Beispiel den etablierten ERP- oder CRM-Anwendungen das Wasser reichen kann. Daher wird die Anwendung nur eine einfache Version dessen implementieren, was man in einer Geschäftsanwendung vorfindet.

Konkret soll es mit dieser Anwendung möglich sein, Bestellungen zu bearbeiten. Dazu muss zum einen der Bestellprozess modelliert werden und zum anderen die davon betroffenen Business-Objekte, nämlich der Kunde, die Waren und schließlich die Bestellung selbst.

1.1 Das fachliche Modell

Das fachliche Modell zeigt Abbildung 1-1. Es gibt die Klasse `Kunde` mit den Attributen `name`, `vorname` und `kontostand`. Diese Attribute sind in den Implementierungsklassen durch `private`-Attribute umgesetzt, die `public`-Zugriffsmethoden haben. Ein weiteres Business-Objekt ist die `Bestellung`, die eine Komposition aus `BestellPositionen` ist. Diese haben jeweils Referenzen auf die `Waren`.

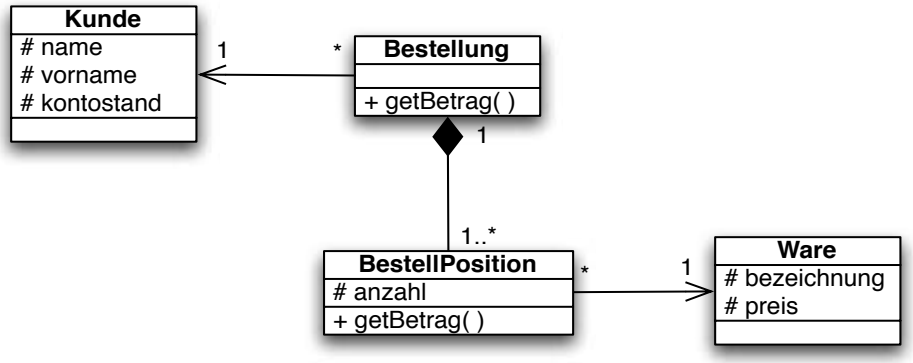

Abb. 1-1 *Fachliches Anwendungsmodell: die fachlichen Klassen Kunden, Waren und Bestellungen*

Die fachlichen Klassen haben kaum eigene Logik. Lediglich die Berechnung des Betrags der `Bestellung` ist hier implementiert. Solche Entwürfe findet man in der Praxis recht häufig: Die Business-Objekt-Schicht dient nur zum Verwalten der Daten und hat wenig echte Logik. Die meiste Logik liegt auf der Ebene der Prozesse und kann daher in den fachlichen Klassen nicht abgebildet werden. Dieser Ansatz wird allerdings auch kritisiert [Fow02] [Eva03], weil man zwar Business-Objekte implementiert, aber die Vorteile von Objektorientierung wie Polymorphie und Vererbung nicht nutzt.

Datenbankzugriff

Die Business-Objekte sollen natürlich dauerhaft in einer Datenbank gespeichert werden. Dazu bekommen die einzelnen Objekte jeweils eine `id` als Ganzzahl, die als eindeutiger technischer Schlüssel dient.

Ein klassisches Problem bei der Entwicklung von Geschäftsanwendungen ist die Frage, wie man die Objekte in der Datenbank ablegt. Die Datenbanken erzwingen meist ein relationales Modell: Die Daten müssen in Tabellen gespeichert werden. Einzelne Datensätze haben einen Wert, der sie eindeutig identifiziert (der *Primärschlüssel*). Referenzen zwischen Objekten müssen also durch Beziehungen zu Primärschlüsseln (so genannten *Fremdschlüsseln*) ersetzt werden.

1.2 Data Access Object (DAO)

Zur Abbildung der Objekte auf die Datenbank-Strukturen wird in der Beispielanwendung das Pattern `Data Access Object` (DAO) verwendet. DAOs kapseln den Zugriff auf die Datenbank vollständig. Sie erlauben es, einzelne Business-Objekte aus der Datenbank zu lesen, sie dort zu erzeugen und zu aktualisieren sowie Listen von Business-Objekten über geeignete Zugriffsmethoden (Abfragen) bereitzustellen.

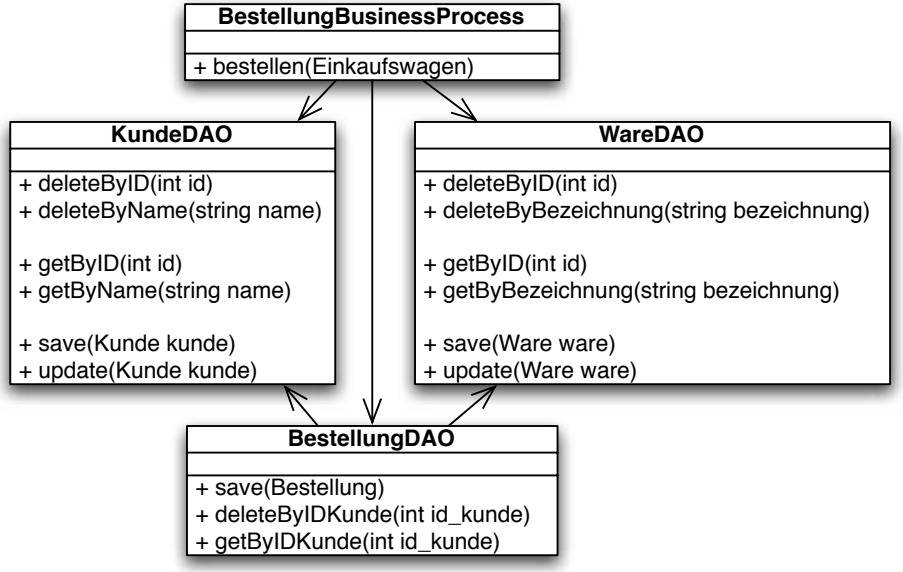

Abb. 1-2 *Die DAOs in der Beispielanwendung*

Die DAOs (Abb. 1-2) sollen möglichst keine Auswirkungen auf die Implementierung der Business-Objekte haben. Die verwalteten Objekte müssen jeweils ein Attribut `id` für den Primärschlüssel haben, aber darüber hinaus nimmt die Implementierung keine Rücksicht auf die Persistenz.

Ein wesentlicher Vorteil ist, dass man durch die Isolation der Persistenz in die DAO-Schicht leichter testen kann: Statt der eigentlichen DAOs kann man Implementierungen für Tests nutzen, die reduzierte Funktionalitäten haben. Dadurch kann man die Anwendung ohne Datenbank testen, was meistens eine erhebliche Vereinfachung ist.

Durch die DAOs sind auch gleich Funktionalitäten implementiert, um neue `Kunden` oder `Waren` anzulegen und sie zu verwalten.

1.3 Geschäftsprozesse in der Beispielanwendung

Die Beispielanwendung soll als Geschäftsprozess einen der wichtigsten Prozesse jedes Unternehmens abbilden: die Bestellung (»Kunde droht mit Auftrag«). Die Schnittstelle des Geschäftsprozesses (Abb. 1-2) verwendet den `Einkaufswagen`. Er enthält Informationen über den Kunden und die bestellten Waren. Dabei werden nur primitive Datentypen verwendet, wie dies bei serviceorientierten Architekturen oder verteilten Systemen oft der Fall ist. So kann man den Dienst verwenden, ohne viel über die dahinter liegenden Business-Objekt-Strukturen zu wissen. Die bestellten Waren und der bestellende Kunde werden also nur durch ihre jeweilige `id` identifiziert. Entsprechend muss der Geschäftsprozess die Business-Objekte zu diesen `ids` zusammensuchen und anschließend die Bestellung erzeugen. Dabei soll noch eine Überprüfung stattfinden, ob der Kunde mit seinem derzeitigen Kontostand die Bestellung überhaupt bezahlen kann.

Um auf die persistenten Daten zugreifen zu können, muss man natürlich den Geschäftsprozess mit Referenzen auf die DAOs ausstatten.

1.4 Benutzeroberfläche

Die Benutzeroberfläche für die Beispielanwendung soll eine Weboberfläche enthalten, die das Anlegen von Daten wie Kunden oder Waren erlaubt. Außerdem muss man natürlich auch die Möglichkeit haben, den Prozess für die Bestellung auszulösen. Diese Funktionalitäten sind mit einigen einfachen Formularen implementiert.

2 Spring

Das Spring-Framework [Spring] ist eine Möglichkeit, die Entwicklung mit Java deutlich zu vereinfachen. Es basiert auf drei Elementen:

1. Spring bietet eine vereinfachte und vereinheitlichte API-Schicht über viele Java-SE-APIs, Java-EE-APIs und Open-Source-Frameworks an. Der wesentliche Grund für die zunehmende Komplexität der Entwicklung mit Java ist nämlich nicht die Sprache selbst, sondern es sind die zahlreichen APIs, die Java anbietet. Diese APIs folgen unterschiedlichen Konzepten und sind oft unnötig schwer zu handhaben. Spring löst dieses Problem.

2. Eine wesentliche Herausforderung bei objektorientierten Systemen ist das Verwalten der Abhängigkeiten zwischen Objekten und zwar insbesondere zwischen jenen, die Geschäftslogik als Services implementieren. Diese Dienste bauen aufeinander auf, so dass man Netze aus solchen Objekten erzeugen muss. Dieses Problem wird oft »irgendwie« im Code gelöst. Spring bietet mit Dependency Injection einen einheitlichen Weg, solche Objektnetze aufzubauen: Den Objekten werden abhängige Objekte anhand einer einfachen XML-Konfiguration zugewiesen (»injiziert«). Die Objekte suchen sich also nicht Referenzen zu anderen Objekten, sondern sind passiv. Dadurch sind sie von der Umgebung unabhängig, da sie diese nicht aktiv benutzen. Sie können flexibel in unterschiedlichen Umgebungen verwendet werden wie z. B. im Applikationsserver oder in einer »normalen« Java-SE-Umgebung. Außerdem ist durch Dependency Injection jede Spring-Anwendung konfigurierbar. Man kann beispielsweise recht einfach eine andere Datenbank nutzen. Vor allem das Testen wird erleichtert, da man das System ohne großen Aufwand in Testumgebungen laufen lassen kann oder den Objekten spezielle Testobjekte mit reduzierter Funktionalität (z. B. Mocks) als abhängige Objekte zuweisen kann.

3. Schließlich bietet Spring eine Unterstützung für AOP (aspektorientierte Programmierung). Normalerweise sind Belange wie Transaktionen, Tracing oder Sicherheit im Code verstreut: Eine Methode eröffnet z. B. eine Transaktion, überprüft, ob der aktuelle Benutzer die nötigen Rechte hat, und implementiert dazwischen die Geschäftslogik. Dieses Vorgehen findet sich in recht vielen Methoden, so dass diese Belange in vielen unterschiedlichen Bereichen des Systems implementiert sind. AOP ermöglicht die zentralisierte und von der Geschäftslogik getrennte Implementierung solcher Belange. Außerdem können durch Aspekte Java-Annotationen recht einfach mit einer auszuführenden Logik verbunden werden.

Ein Ziel von Spring ist, dem Entwickler möglichst viele Freiheiten zu lassen. Man kann daher die einzelnen Teile des Frameworks unabhängig voneinander nutzen. Der Entwickler kann z. B. nur die Teile verwenden, die in seinem Projekt einen besonders großen Vorteil bringen. Außerdem muss man mit Spring nicht eine bestimmte Lösung z. B. für Persistenz verwenden, sondern findet eine breite Palette von in Spring integrierten Frameworks vor. Zudem soll Spring wenig invasiv sein, d. h., der eigene Code hängt nur wenig vom Spring-Framework ab. Eine vollständige Unabhängigkeit lässt sich natürlich nicht immer erreichen.

Durch diese Ansätze richtet sich die Programmierung wieder mehr an Objektorientierung als an den verwendeten APIs aus. So werden Projekte in die Lage versetzt, Lösungen für die jeweiligen Projektziele zu erarbeiten, statt ohne weiteres Nachdenken einem bestimmten Stil zu folgen, der

durch die APIs vorgegeben wird. Gleichzeitig kann man sich mehr auf die eigentliche Logik konzentrieren, statt eigene Abstraktionen über die verschiedenen APIs oder anderen Infrastrukturcode zu entwickeln.

2.1 Dependency Injection mit Spring

In der Spring-Konfigurationsdatei wird das Objektnetz mit XML konfiguriert. Die dort definierten Objekte sind ganz normale Java-Objekte. Man nennt sie auch Spring-Beans, da sie durch Spring erzeugt werden. Die grundlegenden Konfigurationsmöglichkeiten sind:

- Spring-Beans definieren: Man kann Spring-Beans durch das `bean`-Element anlegen. Sie bekommen einen Namen, und man muss definieren, von welcher Klasse die Spring-Beans sein sollen.
- Mit dem `property`-Element kann man einer Property einer Spring-Bean einen Wert zuweisen. Dahinter verbirgt sich der Aufruf einer entsprechenden set-Methode. Damit werden die einzelnen Spring-Beans konfiguriert. Ein Beispiel ist das Setzen des Benutzernamens für eine Datenbankverbindung. Der Wert für die Property wird in einem `value`-Attribut angegeben. Alternativ kann man Referenzen zwischen Objekten herstellen: Dadurch wird das Objektnetz aufgebaut. Hier wird mit dem `ref`-Attribut gearbeitet.

Eine Konfiguration nach diesem Schema findet sich in Listing 2-1. Wie man sieht, gibt es auch ein passendes XML-Schema für die Spring-Konfiguration. Mit Hilfe von XML-Namespaces und eigenen XML-Schemata kann man die Konfiguration erweitern, um so weitere technische Features leicht nutzbar zu machen.

In den Listings werden Package-Namen oft durch … ersetzt, um die Lesbarkeit und die Übersichtlichkeit zu verbessern. Da moderne Entwicklungsumgebungen meistens das Importieren der Klassen aus den entsprechenden Packages automatisieren, muss man den Package-Namen oft auch gar nicht kennen.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
 "http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean name="datasource"
   class="org.apache.commons.dbcp.BasicDataSource"
   destroy-method="close">
    <property name="driverClassName"
     value="org.hsqldb.jdbcDriver" />
    <property name="url"
     value="jdbc:hsqldb:file:springbuchhsqldb" />
    <property name="username"
     value="sa" />
    <property name="password"
     value="" />
  </bean>

  <bean name="kundeDAO" class="springjdbcdao.KundeDAO">
    <property name="datasource" ref="datasource"/>
  </bean>

  <bean name="bestellungDAO"
   class="springjdbcdao.BestellungDAO">
    <property name="datasource" ref="datasource"/>
    <property name="kundeDAO"   ref="kundeDAO"/>
    <property name="wareDAO"    ref="wareDAO"/>
  </bean>
```

```
  <bean name="wareDAO" class="springjdbcdao.WareDAO">
    <property name="datasource" ref="datasource"/>
  </bean>

  <bean name="bestellung"
   class="….BestellungBusinessProcess">
    <property name="bestellungDAO"
     ref="bestellungDAO"/>
    <property name="kundeDAO">
     ref="kundeDAO"/>
    <property name="wareDAO"
     ref="wareDAO"/>
  </bean>

</beans>
```

Listing 2-1 *Eine mögliche Spring-Konfigurationsdatei für die Beispielanwendung*

XML Schema

Bei der Konfiguration der `DataSource` sieht man, wie man einem Objekt mit dem `value`-Attribut direkt Konfigurationswerte zuweisen kann. Den DAOs wird mit dem `ref`-Attribut jeweils eine Referenz auf die `DataSource` zugewiesen, die zum Zugriff auf die Datenbank notwendig ist. Zusätzlich werden die Beziehungen zwischen den DAOs konfiguriert, und schließlich bekommt der Geschäftsprozess alle DAOs zugewiesen.

Zur Auswertung der Spring-Konfiguration kann man einen `ApplicationContext` instanziieren:

```
ApplicationContext applicationContext = new
 ClassPathXmlApplicationContext(
  "beans.xml");
```

Dann kann man aus dem `ApplicationContext` Spring-Beans auslesen:

```
BestellungBusinessProcess bestellung=
 (BestellungBusinessProcess)
  applicationContext.getBean("bestellung");
```

Oft ist es jedoch so, dass man keine Spring-Beans auslesen muss, weil beispielsweise beim Zugriff auf eine bestimmten URL automatisch eine bestimmte Spring-Bean angesprochen wird.

Damit ist also definiert, wie man Objektnetze und Objektkonfigurationen mit Hilfe von Dependency Injection und Spring festlegen kann. Natürlich gibt es deutlich mehr Features in Spring, die man zum Beispiel in [Spring] nachlesen kann. Dort gibt es eine Einführung in AOP, und auch die Unterstützung für die verschiedenen Java-SE-API, Java-EE-APIs und Open-Source-Framework wird erläutert.

3 Was ist OSGi?

Die OSGi Service Platform [OSGi] [WHKL08] ist ein offener Standard, der ein Komponentenmodell auf Basis von Java definiert. Ursprünglich wurde er vor allem in der Embedded-Entwicklung genutzt, aber mittlerweile wird er in vielen verschiedenen Bereichen verwendet. Eine der wichtigsten Anwendungen, die auf OSGi aufsetzen, ist sicher die Eclipse-Plattform die mit Hilfe von OSGi die verschiedenen Plugins verwaltet. Sie hat sich zum Beispiel zur Entwicklung von Rich-Client-Anwendungen etabliert.

Technisch bietet OSGi folgende Features:

- Durch die Modularisierung von Anwendungen in JAR-Dateien oder Verzeichnisse ist festgelegt, wie man einzelne Komponenten deployen und ausliefern kann. Sie entsprechen jeweils einem OSGi-Bundle. Dadurch ist das Deployment-Format eines Bundles definiert und es ist auch klar, dass ein Bundle Java-Packages mit Java-Klassen oder Java-Interfaces enthält. Zudem entspricht dieses Vorgehen dem, wie Java-Anwendungen auch ohne OSGi oft aufgeteilt werden.
- Die Bundles können jedes für sich heruntergefahren und neu gestartet werden, man kann sie einzeln updaten oder installieren. Diese Features verbessern die Verfügbarkeit von Anwendungen, weil sie nicht vollständig ausfallen, wenn nur einzelne Teile per Update aktualisiert oder neu gestartet werden müssen. Bei klassischen WAR- oder gar EAR-Dateien, wie man sie bei Java-EE-Anwendungen nutzt, müsste man hingegen selbst bei kleinen Änderungen die gesamte Anwendung neu deployen. Außerdem kann man mit OSGi sicherstellen, dass nur ein bestimmtes Bundle geändert wird. (Im Java-EE-Kontext muss dagegen meistens die gesamte WAR- oder EAR-Datei, die den gesamten Code enthält, neu erzeugt werden, denn selbst wenn man sie nur anpasst, kann man nicht ausschließen, dass man nicht aus Versehen irgendwo unbeabsichtigte Änderungen vorgenommen hat.) So kann man mit OSGi beispielsweise Regressionstests vereinfachen: Bundles, die sich nicht geändert haben, müssen weniger intensiv getestet werden, während man ohne OSGi eben auch die eigentlich unveränderten Teile sicherheitshalber testen muss – es könnte sich ja aus Versehen doch ein Unterschied eingeschlichen haben.
- OSGi ist außerdem recht flexibel: Man kann in einer OSGi-Runtime z. B. auch durchaus einen Webserver laufen lassen. Zum Administrieren bietet eine OSGi-Implementierung eine geeignete Benutzeroberfläche. Die OSGi-Implementierung Equinox [Equinox], auf der Eclipse basiert, hat z. B. eine Konsole, in der man Befehle für diese Funktionen eingeben kann. Bei der Knopflerfish-OSGi-Implementierung gibt es die Knopflerfish-Konsole, mit der man auch entfernte Rechner mit einer OSGi-Umgebung administrieren kann [Knopflerfish].
- Die Bundles können Java-Packages importieren oder exportieren. Alle anderen sind privat für das Bundle. Auch kann ein Bundle ein ganzes anderes Bundle importieren. Durch Eingriffe in die Class-Loading-Mechanismen von Java ist es ausgeschlossen, dass ein anderes Bundle auf die privaten Klassen eines Bundles zugreift. Das Konzept der Unterscheidung in öffentliche und private Anteile ist auf Ebene der Java-Klassen mit ihren Instanzvariablen und Methoden ebenfalls vorhanden, aber nur durch die Ausweitung auf Packages und ihre Klassen kann man auch größere System sinnvoll modularisieren. Die notwendigen Einstellungen werden in der Datei `/META-INF/MANIFEST.MF` festgelegt, die jede JAR-Datei sowieso enthält. Listing 3-1 zeigt ein Beispiel.

- Es gibt in OSGi eine Service-Registry, in die sich Services registrieren können, die dann anderen Bundles zur Verfügung stehen. Diese Services werden durch Java-Objekte implementiert. Die Services müssen keine speziellen OSGi-Interfaces implementieren und sind damit sehr leichtgewichtig und einfach testbar.
- Java-Klassen haben keine Versionen. OSGi definiert für die exportierten und importierten Packages und für die Bundles ein Versionskonzept, mit dem dieses Feature zur Java-Plattform hinzugefügt wird. Dadurch ist z. B. auch eine schrittweise Migration zu einer neuen Version möglich, indem man das Bundle mit der alten Version noch so lange laufen lässt, wie es noch genutzt wird. Parallel kann schon die neue Version verwendet werden. Nur mit einem solchen Versionierungskonzept kann man größere Softwaresysteme wirklich änderbar und dadurch wartbar machen.

```
Bundle-Version: 1.0
Bundle-SymbolicName: de.springbuch.springbuchosgi
Bundle-Name: de.springbuch.springbuchosgi
Export-Package: de.springbuch;version:=2.0.0
Import-Package: org.apache.log4j, org.apache.log4j.xml
```

Listing 3-1 *Beispiel für ein MANIFEST.MF aus einem OSGi-Bundle*

Listing 3-1 zeigt auch schon die Versionen: In der vierten Zeile wird definiert, dass das Bundle das Package `de.springbuch` exportiert und zwar in der Version 2.0.0 . Die Imports sind nicht versioniert, es würden also jetzt beliebige Versionen irgendeines Bundles aus der OSGi-Umgebung verwendet werden. Das ist normalerweise keine gute Idee, da die Anwendung wahrscheinlich nur mit einer bestimmten Version funktionieren kann. Also könnte man zum Beispiel `org.apache.log4j;version=1.2.15` als Import nutzen. Der Nachteil dieses Imports ist allerdings, dass es nur definiert, dass man mindestens 1.2.15 nutzen will. Eine Version 3.0.0, wenn es sie denn gibt, würde also einfach verwendet – das ist wahrscheinlich aber nicht gewollt. Um die Imports weiter einzuschränken, kann man eine Schreibweise nutzen, die durch die Intervalle in der Mathematik beeinflusst ist:

- `org.apache.log4j;version=[1.2.15,1.2.15]` würde genau die Version 1.2.15 importieren und sonst keine andere.
- `org.apache.log4j;version=[1.2.15,1.3.0)` würde mindestens Version 1.2.15 wählen und höchstens 1.3.0. Die Version 1.3.0 selber ist aber nicht eingeschlossen.
- `org.apache.log4j;version=[1.2.15,2.0.0)` würde mindestens 1.2.15 wählen und höchstens 2.0.0. Auch hier ist die Version 2.0.0 selber wieder ausgeschlossen. Damit wäre 1.4.0 beispielsweise eine mögliche Version.

OSGi stellt also ein Komponentenmodell zur Verfügung, das sich an grobgranularen Bundles als Komponenten orientiert, die aus Packages und Klassen bestehen. Spring hat ebenfalls ein Komponentenmodell, das aber wesentlich feingranularer ist, weil es sich an Java-Klassen orientiert. Daher können sich Spring und OSGi gut ergänzen: Man baut die Anwendung auf Basis des grobgranularen OSGi-Komponenten-Modells auf, und die OSGi-Bundles verwenden intern Spring zur Strukturierung der Bundles. Die Bundles können dann untereinander »verdrahtet« werden, indem Services zu Verfügung gestellt werden bzw. genutzt werden. Dadurch stellen die Bundles ihre Funktionalitäten anderen Systemen durch die Services zur Verfügung. Die Services bilden sozusagen die Schnittstellen der Bundles.

Außerdem kann man natürlich andere Feature von Spring wie die Unterstützung für die verschiedenen Java-SE-APIs, Java-EE-APIs und Open-Source-Frameworks innerhalb einer OSGi-Anwendung verwenden. Außerdem hilft OSGi auch bei der Definition von Sichtbarkeiten für Typen

wie oben erläutert. Dieser Bereich wird von Spring ebenfalls nicht unterstützt.

Spring selbst wird übrigens schon als Sammlung von OSGi-Bundles ausgeliefert. Man kann sich daher bei der Nutzung von Spring die Versionierung wie dargestellt zu Nutze machen. Also kann man die Anwendung auf einer OSGi-Implementierung laufen lassen und in dieser Anwendung dann Spring nutzen.

4 Spring Dynamic Modules for OSGi Service Platforms

Eine weitergehende Integration zwischen Spring und OSGi wird im Projekt *Spring Dynamic Modules for the OSGi Platform* realisiert [SpringOSGi]. Zunächst kann man mit dieser Technologie innerhalb eines OSGi-Bundles einen Spring-`ApplicationContext` initialisieren. Dazu kann man in dem OSGi-Bundle – das ja letztendlich nur eine JAR-Datei oder ein Verzeichnis ist – im Verzeichnis `/META-INF/spring` Dateien mit der Extension `.xml` anlegen. Diese Dateien werden beim Aufbau des `ApplicationContext` im OSGi-Bundle automatisch ausgewertet. Genau genommen ist das Erzeugen des `ApplicationContext` eine Aufgabe des `org.springframework.osgi.extender`-Bundles. Um also einen Spring-`ApplicationContext` in einem OSGi-Bundle verwenden zu können, muss man lediglich dieses OSGi-Extender-Bundle installieren und dann die Spring-Konfigurationen in dem dafür vorgesehenen Verzeichnis ablegen – der Rest funktioniert automatisch. Im dm Server ist diese Infrastruktur selbstverständlich bereits enthalten.

Offen ist die Frage, wie man mit Spring OSGi-Services definiert, die die Schnittstelle eines Bundles nach außen darstellen. Dazu kann man prinzipiell OSGi-Services manuell exportieren und nutzen. Dies hat jedoch einige Nachteile:

- Der Code hängt von OSGi ab, was eine Wiederverwendung in einem anderen Kontext – zum Beispiel Java EE – unmöglich macht. Da Zugriffe auf Services im Code recht häufig notwendig sind, handelt man sich leicht Abhängigkeiten auf breiter Front ein.
- Bundles und damit Services können zur Laufzeit installiert werden oder deinstalliert werden und somit verschwinden. Das muss man bei der Programmierung beachten. Man muss also vor dem Zugriff auf einen Service ermitteln, ob er überhaupt zur Verfügung steht. Das macht den Code komplexer.
- Außerdem muss man dafür sorgen, dass OSGi-Services deregistriert werden, wenn sie nicht mehr zur Verfügung stehen. Auch hier wird der Code also komplexer.

Daher wäre es gut, wenn man OSGi-Services nicht mit den OSGi-APIs implementieren müsste. Spring Dynamic Modules bietet dafür eine Lösung: Innerhalb eines `ApplicationContext` kann man Spring-Beans als OSGi-Services exportieren. Dazu werden Exporter für den Export eines Service sowie Proxies zum Ansprechen eines Service genutzt. Das ist genau dasselbe Prinzip, wie Spring es für JMX und verschiedene Netzwerkprotokolle verwendet. Bei OSGi wird für die Konfiguration von Exporter und Proxy ein eigener XML-Namespace verwendet. Eine beispielhafte Konfiguration zeigt Listing 4-1. Es wird der `osgi`-Namespace verwendet. Die einzelnen Elemente sind durch das `osgi`-Präfix zu erkennen.

Durch das `osgi:service`-Element wird festgelegt, welcher Service exportiert werden soll. In diesem Fall wird die Spring-Bean `bestellung`, wie sie in Listing 2-1 definiert wurde, als OSGi-Service exportiert, der dann die Methoden aus dem Interface `businessprocess.IBestellungBusinessProcess` zur Verfügung stellt.

Man kann so also eine Anwendung in verschiedene Bundles aufteilen und definieren, welche Spring-Beans zu OSGi-Services werden sollen. Sie legen die Schnittstelle des Bundles nach außen fest. Dabei wird im Code nirgendwo eine OSGi-API genutzt: Nur durch die Deklaration in der Spring-Konfiguration werden die Spring-Beans zu OSGi-Services.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <osgi:service id="bestellenServiceOsgi"
   ref="bestellung"
   interface="businessprocess.IBestellungBusinessProcess"
  />

</beans>
```

Listing 4-1 *Export einer Spring-Bean als OSGi-Service*

Es muss natürlich auch eine Möglichkeit geben, OSGi-Services von anderen Bundles zu importieren. Dazu gibt es im `osgi`-Namespace das `osgi:reference`-Element und das `osgi:list`-Element, falls man Referenzen auf mehrere Services haben will. Listing 4-2 zeigt ein Beispiel: Man definiert jeweils das benötigte Interface in diesen Elementen und erhält dann eine Referenz auf das gewünschte Interface, wie im Beispiel bei der Spring-Bean `bestellung`, bzw. eine Liste solcher Instanzen, wie bei der Spring-Bean `listeners`. Diese Referenzen auf OSGi-Services kann man wie alle anderen Spring-Beans auch nutzen. Die Verwendung von OSGi ist also transparent: Weder das Bundle, das den Service anbietet, noch das Bundle, das den Service nutzt, hängt in irgendeiner Weise von OSGi ab.

```
<osgi:reference id="bestellung"
 interface="businessprocess.IBestellungBusinessProcess"
 timeout="3000"
/>
```

```
<osgi:list id="listeners"
 interface="EventListener"
/>

<bean id="myBean" class="SomeClass">
  <property name="bestellen" ref="messageService"/>
  <property name="eventListeners" ref="listeners"/>
</bean>
```

Listing 4-2 *Beispiel für den Import eines OSGi-Service*

Services unterscheiden sich von Spring-Beans nicht nur durch die Kardinalität, sondern auch dadurch, dass sie heruntergefahren werden können. Normalerweise muss man mit diesem Umstand selber programmatisch umgehen. Bei Dynamic Modules wird einem diese Arbeit abgenommen: Wird der Service heruntergefahren und dann trotzdem aufgerufen, wird der Methodenaufruf von Spring gepuffert. Ist der Service rechtzeitig wieder verfügbar, wird die Methode ausgeführt. Mit dem `timeout`-Attribut, wie es in Listing 2-1 für die `bestellung`-Bean definiert wurde, kann man festlegen, wie lange Dynamic Modules darauf wartet, dass der Service wieder zur Verfügung steht, bevor eine Exception geworfen wird. Bei einer Liste wird die entsprechende Instanz einfach automatisch aus der Liste entfernt.

So kann ein Update von einzelnen Bundles vorgenommen werden, ohne dass man die ganze Anwendung stoppen müsste. Es geht dabei kein einziger Methodenaufruf verloren, da diese ja gepuffert werden.

Manchmal reicht ein Automatismus aber nicht aus. Daher gibt es auch die Möglichkeit, sich beim Ausfall des Services informieren zu lassen. So kann man mit `osgi:listener` eine Spring-Bean registrieren, die beim Ausfall des Service aufgerufen wird. Sie muss entweder das Interface `OsgiServiceRegistrationListener` implementieren, das Methoden definiert, die beim Ausfall und beim erneuten Starten des Services aufgerufen werden.

Alternativ kann man auch mit dem `bind-method`-Attribut und dem `unbind-method`-Attribut die Namen der Methoden festlegen, die aufgerufen werden sollen. (Listing 4-3). Dies vermeidet, dass der Code von einem Spring-OSGi-spezifischen Interface abhängt.

```
<osgi:reference id="bestellenService"
 interface="de.springbuch.IBestellungBusinessProcess" >
  <osgi:listener ref="bestellenAusgefallenBean"/>
  <osgi:listener ref="bestellenAusgefallenBeanPojo"
    bind-method="serviceAusgefallen"
    unbind-method="serviceWiederDa"/>
</osgi:reference>
```

Listing 4-3 *Listener für das Hochfahren und Runterfahren von Services*

Damit sind die wesentlichen Funktionen von Dynamic Modules erläutert. Darüber hinaus gibt es noch einige interessante Funktionen:

- Durch die `AbstractConfigurableBundleCreatorTests` kann man Spring-OSGi-Bundles in einer JUnit-Umgebung testen, ohne dabei einen OSGi-Container konfigurieren zu müssen.
- Zugriffe auf Ressourcen aus dem OSGi-Container sind möglich. Dazu kann man eine Ressource mit dem `bundle:`-Präfix versehen.
- Zugriff auf den `BundleContext` ist für jede Spring-Bean möglich, die `BundleContextAware` implementiert. Dadurch bekommt man umfangreichen Zugriff auf die Features des OSGi-Containers.
- Es gibt eine Schnittstelle, um mit dem OSGi-Configuration-Admin-Service auch Einstellung für Spring-Beans verwalten zu können.

Spring und OSGi ergänzen sich also recht gut. Vor allem fügt OSGi zu Spring Features hinzu, die für den zuverlässigen Betrieb von Anwendungen notwendig sind, da man die Bundles einzeln starten, stoppen und updaten kann. Außerdem wird durch die Bundles möglich, Anwendungen weiter

zu modularisieren. Und natürlich kann man innerhalb einer OSGi-Anwendung die Integration der verschiedenen APIs nutzen die Spring anbietet – z. B. im Bereich Transaktionen, Persistenz oder Remoting.

5 dm Server

Spring Dynamic Modules for the OSGi Platform bietet eine grundlegende Integration zwischen Spring und OSGi. Um OSGi aber ernsthaft für Enterprise-Anwendungen zu nutzen, ist mehr notwendig. Mit dem dm Server steht ein Applicationsserver zur Verfügung, der es recht einfach macht, mit OSGi Enterprise-Anwendungen zu entwickeln. Er ist ein Open-Source-Projekt und steht unter der GPL-Lizenz. Außerdem gibt es eine Version mit einer kommerziellen Lizenz und zusätzlichen Features für die Administration und das Monitoring der Anwendungen. So kann diese Version die Performance der Anwendung überwachen und zwar auf der Ebene der einzelnen Spring Beans.

Web Modules

Die erste Erweiterung des dm Servers gegenüber einfachem OSGi sind die sogenannten Web Modules. Normalerweise enthält OSGi nur einfache Bundles, die Klassen enthalten. Im Enterprise-Java-Bereich haben sich mittlerweile Dateiformate etabliert, die bestimmte Arten von Anwendungen enthalten. Dazu zählen z. B. die WAR-Dateien für Webanwendungen. Der dm Server erlaubt die Nutzung spezifischer OSGi-Bundle für bestimmte Einsatzzwecke wie Webanwendungen. Dadurch wird auch in diesem Bereich die Modularisierung in Bundles genutzt, aber gleichzeitig kann man die spezifischen Anwendungsszenarien besser unterstützen.

Dazu sind einige Erweiterungen zu den sonst üblichen OSGi-Bundles definiert:

- Ein Web-Bundle enthält im Verzeichnis `/MODULE-INF` die notwendigen Ressourcen wie JSPs. In normalen OSGi-Bundles gibt es für solche Ressourcen keine gesonderten Verzeichnisse. Man kann in dieses Verzeichnis auch eine `WEB-INF/web.xml` hinzufügen, also eine sonst für Webanwendungen übliche Konfigurationsdatei. Die Inhalte dieser Datei wird mit den übrigen Einstellungen aus `/META-INF/MANIFEST.MF` kombiniert. Dadurch ist es möglich, auch kompliziertere Konfigurationen für die Webanwendung vorzunehmen und zwar so, wie man es von normalen Webanwendungen gewohnt ist.

- Es wird bereits ein `DispatcherServlet` konfiguriert. Dies ist das grundlegende Element, um Spring-Beans in einer Webumgebung anzusprechen. Die im Verzeichnis `/META-INF/spring` vorhandenen Spring-Konfigurationsdateien werden ausgewertet, wie dies auch bei Dynamic Modules der Fall ist. Aber bei einem Web-Bundle stehen auch alle Controller aus diesen Dateien stehen automatisch in der Webanwendung zur Verfügung. Controller sind solche Spring-Beans, die direkt auf Web-Requests reagieren. Das vereinfacht die Konfiguration einer Spring-Webanwendung sehr nachhaltig.

- In der `/META-INF/MANIFEST.MF`-Datei werden einige zusätzliche Einstellungen vorgenommen. Listing 5-1 zeigt ein Beispiel: Durch `Module-Type` wird festgelegt, das dieses Bundle ein Web-Bundle ist. Hier ist für spätere Versionen des dm Servers die Unterstützung für andere Bundle-Typen zum Beispiel für Batch-Verarbeitung oder Integration denkbar. Anschließend wird mit `Web-DispatcherServletUrlPatterns` definiert, auf welche URLs das `DispatcherServlet` reagieren soll. Und der `Web-ContextPath` bestimmt, unter welcher URL die Anwendung zur Verfügung stehen soll. Im Beispiel steht also das gesamte Web-Bundle unter `/spring-buch` zur Verfügung. Wenn im `MODULE-INF`-Unterverzeichnis nun eine Datei `index.html` steht, kann man auf sie unter `http://localhost/spring-buch/index.html` zugreifen. Auf das `DispatcherServlet` kann man unter `http://localhost/spring-buch/springbuchweb` zugreifen.

```
Manifest-Version: 1.0
Bundle-Name: Web Bundle
Bundle-ManifestVersion: 2
Bundle-SymbolicName: de.spring_buch.web
Bundle-Version: 1.0.0
Module-Type: Web
Web-DispatcherServletUrlPatterns: /springbuchweb/*
Web-ContextPath: /spring-buch
...
```

Listing 5-1 */META-INF/MANIFEST.MF-Datei eines Web-Bundles*

Wie man sieht, ist es also sehr einfach möglich, eine Webanwendung als Web-Bundle zu implementieren. Vor allem ist die Spring-Infrastruktur schon passend konfiguriert, so dass man wesentlich weniger Einstellungen vornehmen muss.

Dennoch sind die meisten Webanwendungen heutzutage in WAR-Dateien abgelegt, und diese will man nicht vollständig umstellen. Der dm Server kann auch mit WAR-Dateien umgehen. Diese werden beim Deployment intern ebenfalls zu Web-Bundles. Dadurch kann man sehr einfach mit dem dm Server starten, weil man im ersten Schritt einfach nur seine Webanwendungen als WARs auf einer anderen Plattform deployt.

WAR zu Web-Bundle

Dennoch kann man dann noch nicht die Vorteile von OSGi ausnutzen:

- Die WAR-Dateien enthalten alle Libraries, was einen Austausch einer Library gegen eine neue Version schwierig macht – insbesondere wenn mehrere Anwendungen die Library nutzen. Und auch das Versionierungssystem von OSGi kann nicht genutzt werden.

- Auch sonst ist die WAR-Datei monolithisch: Sie enthält nicht nur den Webanteil der Anwendung, sondern alles – auch die Logik. Egal, wie stark die Anwendung intern modularisiert ist: Auf Ebene der WAR-Datei wird es einfach ein einziges, großes Modul. Dadurch ergeben sich einige Probleme, die nicht trivial lösbar sind. So kann man nicht ohne Weiteres von einer WAR-Datei auf die Logik in einer anderen WAR-Datei zugreifen. Eine Lösung wäre es, beide in eine EAR-Datei zu verpacken. Dann muss man aber beide und auch den restlichen Inhalt der EAR-Datei immer zusammen deployen. Im Extremfall reicht also eine Layout-Änderung, damit man eine komplette Anwendung und eine Menge Elemente, die gar nicht beeinflusst worden sind, neu deployen muss – mit entsprechender Downtime. Und man muss die Anwendung auch regressionstesten, um auszuschließen, dass irgendwo etwas fälschlicherweise geändert wurde. Abgesehen davon, dass einfache Webserver, die immer häufiger anstelle von Applikationsservern genutzt werden, EARs oft gar nicht unterstützen. Eine andere Möglichkeit ist es, die Logik in Form von Web Services zur Verfügung zu stellen, so dass die beiden WARs über HTTP miteinander kommunizieren können. Das ist dann aber kein rein lokaler Zugriff mehr. Dadurch muss man mit einen größeren Performance-Overhead rechnen, und man kann auch nicht mehr so einfach Transaktionen definieren, die solche Dienste miteinbeziehen. Denn dann würden die Informationen über die Transaktion per HTTP mittransportiert werden, was nicht einfach ist und auch meistens nicht gewünscht wird.

Der dm Server bietet den Vorteil, dass die WAR-Dateien intern in ein OSGi-Web-Bundle umgewandelt werden. Daher kann man mit diesem Server die Vorteile von OSGi nutzen, ohne dass man gleich die WAR-Dateien gegen ein völlig anderes Format austauschen müsste:

Shared Libraries WAR

Als Erstes kann man in der WAR-Datei ein `/META-INF/MANIFEST.MF` unterbringen, in dem man dann OSGi-Header verwenden kann. Dadurch kann man die Libraries aus dem dm Server laden, statt sie mit in der WAR-Datei auszuliefern. So werden die WARs kleiner und man kann außerdem von der OSGi-Versionsverwaltung profitieren. Man kann nämlich recht einfach eine neue Version einer Library einspielen, die dann von allen Anwendungen genutzt wird. Das Vorgehen entspricht der Idee, dass die Infrastruktur einer Anwendung vom Applikationsserver bereit gestellt wird – bei klassischen WAR-Dateien ist die Infrastruktur meistens in Form von Libraries in der Anwendung enthalten. Das Vorgehen, die Libraries unter die Kontrolle des dm Servers zu stellen, nennt man auch »Shared Libraries WARs«. Diese WARs nutzen zwar noch das Layout eines normalen WAR-Files, aber intern OSGi für den Zugriff auf die Libraries.

Shared Services Bei Shared Library WARs ist die Logik immer noch direkt in der WAR-Datei enthalten. Im dm Server kann man aber in der Spring-Konfiguration einer WAR-Datei die `osgi`-Elemente verwenden, um OSGi-Services anzusprechen. Dann kann man die Services in eigene Bundles auslagern und als OSGi-Services ansprechbar machen. Dadurch sind die Vorteile wie die Nutzung dynamischer Services und die Möglichkeit zur Pufferung von Methodenaufrufen auch mit normalen WAR-Dateien nutzbar. Wie man sieht, muss man also nicht unbedingt auf Web-Bundles umsteigen, um die Vorteile von OSGi zu nutzen, sondern man kann schrittweise bestimmte Features von OSGi auch mit traditionellen WAR-Dateien nutzen. Man spricht von einem Shared Services WAR.

Durch die Nutzung dieser Erweiterungen für das WAR-Format ist es möglich, wesentliche Vorteile von OSGi zu nutzen, ohne dass man seine Anwendung vollständig umstellen muss. Insbesondere können Build-Prozesse beibehalten werden, die für die Erzeugung der WARs genutzt werden. Dadurch wird auch der Umstieg auf OSGi oder den dm Server vereinfacht:

Man muss nicht die komplette Anwendung auf OSGi umstellen, sondern kann zunächst die vorhandenen WAR-Dateien deployen, dann die Behandlung der Libraries ändern und schließlich auch OSGi-Services nutzen.

OSGi-Anwendungen

Ein weiteres Problem, dass der dm Server löst, ist der fehlende Applikationsbegriff in OSGi. OSGi kennt nur Bundles, und mehrere Bundles können sozusagen zufällig zusammen die Funktionen einer Anwendung erbringen. Durch den fehlenden Begriff der Anwendung ergeben sich einige Probleme, wenn man versucht, mehrere Anwendungen in einer OSGi-Umgebung laufen zu lassen:

- Typen und Services können von beliebigen Bundles verwendet werden. Nehmen wir an, dass eine Anwendung zur Adressverwaltung und eine Anwendung zur Bestellabwicklung in demselben OSGi-Container laufen sollen. Beide enthalten ein Bundle mit dem Package `business-objects` in der Version 1.0 und darin eine Klasse `Kunde`. Wenn nun ein anderes Bundle dieses Package importiert, bekommt es zufällig eines der Packages aus den beiden Bundles. Nun kann man argumentieren, dass man diese Situation vermeiden kann, wenn man solche doppelten Namen verhindert. Aber dann hängen Anwendungen indirekt voneinander ab, weil sie bei der Benennung von Packages Rücksicht aufeinander nehmen müssen.
- Noch schlimmer ist die Situation bei OSGi-Services. Beispiele dafür sind `DataSources`, also Pools von Datenbankverbindungen, typischerweise OSGi-Services. Auch bei den Services kennt OSGi keine Einschränkung auf eine Applikation, so dass eine Applikation auf die `DataSource` einer anderen Applikation zugreifen könnte – was sicher nicht gewünscht ist und zu merkwürdigen Fehlersituationen führen kann.
- Man kann auch nicht die Applikation zweimal gleichzeitig deployen, da es nicht erlaubt ist, Bundles mit demselben Namen und derselben Version mehrfach zu deployen.
- Schließlich sollte es eine Möglichkeit geben, eine Anwendung in eine einzige Datei zu verpacken. Das ist aber leider auch nicht so einfach, denn OSGi kennt nur Bundles, also JAR-Dateien, aber kein Konzept, um mehrere zusammengehörige Bundles gemeinsam zu deployen. Um also eine Applikation zu deployen, muss man mehrere Bundles installieren, was komplex und zeitaufwändig ist.

PAR-Dateien

Aus diesem Grund führt der dm Server PAR-Dateien ein. Eine PAR-Datei ist eine JAR-Datei, die OSGi-Bundles enthält und eine Anwendung repräsentiert. Damit kann man eine ganze Anwendung als eine einzige Datei deployen. Außerdem definiert eine solche Anwendung einen Sichtbarkeitsbereich für Services und Typen, so dass die oben genannten Probleme gelöst sind. Wenn also ein Bundle innerhalb einer PAR-Datei einen Service oder ein Package anbietet, ist dies nur innerhalb desselben PARs sichtbar. Natürlich kann man von innerhalb des PARs weiterhin auf Packages und Bundles zugreifen, die außerhalb des PARs liegen.

JPA und Hibernate mit OSGi

Auch ein anderes Problem lösen die PARs: Die Nutzung von Enterprise-Libraries wie JPA oder Hibernate in einer OSGi-Umgebung sind normalerweise nur schwer zu realisieren. Ein Grund dafür ist, dass die Persistenzlösung Geschäftsobjekte ändern muss, um Lazy Loading zu implementieren. Wenn man von der Bestellung zum Kunden navigiert, kann es gut sein, dass man den Kunden erst noch aus der Datenbank nachladen muss. Um das zu ermöglichen, muss die Implementierung der Bestellung von der Persistenzlösung so geändert werden, dass sie dieses Nachladen

zusätzlich implementiert. Außerdem muss das nun geänderte Geschäftsobjekt auch Zugriff auf die Persistenzlösung haben, also auch die Klassen der JPA-Implementierung sehen können, um auf die Datenbank zuzugreifen. In einer OSGi-Umgebung treten nun mehrere Probleme auf:

- Die Änderung der Geschäftsobjekt-Klassen ist so einfach nicht möglich, da die Persistenzlösung nicht die Packages der Geschäftsobjekte importiert. Das wäre auch nicht sinnvoll, da die Persistenzlösung ja nicht alle potentiellen Geschäftsobjekte importieren kann.
- Ebenfalls müssen alle Bundles, die auf die Geschäftsobjekte zugreifen, auch die Persistenzlösung importieren. Das sollte jedoch automatisch geschehen, da man manuell dafür nur schwer sorgen kann. Außerdem sollten andere Bundles kein Wissen darüber haben, welche Persistenzlösung in der Anwendung verwendet wird und daher die Persistenzlösung nicht kennen.

Analoge Probleme gibt es, wenn man mit AspectJ Load Time Weaving Klassen in einem OSGi-Umfeld modifizieren will. Dabei werden Klassen nachträglich durch Änderungen am Byte-Code um Features erweitert. Auch hier müssen die Klassen AspectJ bekannt sein, und alle Nutzer der geänderten Klassen müssen auf AspectJ zugreifen können.

Der Begriff der Anwendung, der durch den dm Server in den OSGi-Bereich eingeführt wird, kann auch diese Probleme lösen. Man kann die Persistenzlösung alle Klassen in einer Anwendung modifizieren lassen. Mit diesen Mechanismen ist es recht einfach möglich, sowohl AspectJ Load Time Weaving als auch O/R-Mapper in einem OSGi-Kontext einzusetzen. So kann das Nachladen implementiert werden. Außerdem kann man allen Bundles in einer Anwendung bestimmte Klassen zur Verfügung stellen, ohne dass sie diese importieren müssen. Dadurch kann die Persistenzlösung überall in der Anwendung zur Verfügung stehen.

OSGi-Imports mit dm Server

Man kann nämlich bei einem Import im `/META-INF/MANIFEST.MF` durch ein angehängtes `import-scope=appli cation` die importierten Klassen in der gesamten Anwendung verwendbar machen, so dass Hibernate oder JPA global im ganzen PAR zur Verfügung stehen. Das Bundle mit der Implementierung der Persistenz kann also die Persistenz-Bibliothek mit dieser Sichtbarkeit importieren und so der gesamten Anwendung zur Verfügung stellen.

Außerdem kann man mit dem `context:load-time-weaver`-Element innerhalb des dm Servers AspectJ Load Time Weaving aktivieren. Dazu sind keine weiteren Modifikation am dm Server oder Ähnliches notwendig. Das Load Time Weaving bezieht sich dann auf die gesamte Anwendung, also den kompletten Inhalt des PARs. Es könnten alle Klassen innerhalb der gesamten Anwendung geändert werden. Wenn man in der Spring-Konfiguration einen JPA-`EntityManager` konfiguriert, nutzt dieser automatisch denselben Load Time Weaver und kann so ebenfalls alle Klassen aus der Anwendung manipulieren.

Übrigens biete der dm Server noch einige andere Vorteile im Bereich Import an:

- Mit dem `Import-Library`-Element in der `/META-INF/MANIFEST.MF`-Datei kann man mit einem einzigen Eintrag eine ganze Reihe von Bundles importieren. Dadurch ist es möglich, eine Bibliothek wie Spring, die aus vielen Bundles besteht, mit nur einem Import vollständig zu nutzen.
- Außerdem führt dm Server ein `Import-Bundle`-Element ein. Dadurch kann man alle Packages eines Bundles auf einen Schlag importieren.

Beide Elemente werden intern in `Import-Package`-Elemente umgewandelt. Die OSGi-Implementierung musste also dafür nicht modifiziert werden. Außerdem hat dieses Vorgehen beispielsweise gegenüber `Require-Bundle`

einige Vorteile, wie [WHKL08] diskutiert. `Require-Bundle` ist eine andere Alternative, alle Packages eines bestimmten Bundles nutzbar zu machen.

Der dm Server hat also zum einen den Vorteil, dass klassische Webanwendungen in WAR-Dateien einfach in dem dm Server deployt werden können und man kann sie dann schrittweise mit der Nutzung von OSGi-Features erweitern. Außerdem werden viele typische Probleme bei der Nutzung von OSGi für Enterprise-Anwendungen gelöst. So ist die Nutzung von typischen Persistenzlösungen wie Hibernate oder JPA in einer OSGi-Umgebung sonst nicht sehr trivial, mit dem dm Server ist es kein Problem. Analog wird durch die Unterstützung des Anwendungsbegriffs auch einige Dinge wesentlich vereinfacht.

6 Umsetzung der Beispielanwendung

Die in Kapitel 2 beschriebene Beispielanwendung kann man mit Hilfe des dm Servers und OSGi sehr gut modularisieren. Wie das aussehen kann, stellen wir im Folgenden ausführlich vor. Der Code steht unter `http://spring-buch.de` zum Download zur Verfügung.

Zur Modularisierung wird folgende Aufteilung gewählt:

- In einem Infrastruktur-Bundle werden grundlegende technische Services wie eine `DataSource` zum Zugriff auf die Datenbank definiert und den anderen Bundles zur Verfügung gestellt.
- Ein API-Bundle enthält die Interfaces der DAOs und Business-Prozesse sowie die dazugehörigen Klassen wie Geschäftsobjekte.
- In einem JPA-DAO-Bundle werden die DAO-Interfaces mit einer konkreten Technologie, in diesem Fall JPA, implementiert.
- Das Business-Prozess-Bundle enthält die Implementierung der Business-Prozesse.
- Und schließlich gibt es ein Bundle für die Weboberfläche. Man hätte für die Weboberfläche auch eine WAR-Datei nutzen können, aber wenn man sowieso eine komplett neue Technologie nutzt, kann man auch gleich auf Web Bundles statt auf klassische WAR-Dateien setzten. Bei der Migration einer vorhandenen Anwendung könnte diese Entscheidung anders ausfallen.

Dieser Aufteilung liegen folgende Prinzipien zugrunde:

- Um zur Laufzeit den Austausch der Implementierungen wirklich vornehmen zu können, muss man die Implementierung von den Interfaces trennen. Wenn man dies nicht tut, würde bei einem Austausch nicht nur die Implementierung, sondern auch das Interface ungültig werden. Dann müssen die Bundles, die Interfaces oder Implementierung nutzen, bei einem Austausch der Implementierung neu gestartet werden. Durch die in diesem Beispiel genutzte Aufteilung nutzen andere Bundles aber nur noch die Interfaces. Weil sie in einem anderen Bundle liegen, sind die Interfaces bei einem Austausch der Implementierung nicht beeinträchtigt, so dass man nun problemlos die Implementierung austauschen kann. Im vorliegenden Beispiel sind alle Interfaces in einem einzigen Bundle konzentriert. Im Allgemeinen kann man sie auch in mehrere Bundles aufteilen, damit die Bundles nicht allzu groß werden.
- Um die Anwendung nicht allzu unübersichtlich werden zu lassen, werden die Implementierungen in die einzelnen technischen Schichten (Business-Prozesse, DAOs und Web) unterteilt. Bei größeren Projekten kann eine zusätzliche Aufteilung nach fachlichen Konzepten wie Bestellung, Kunde oder Ware sinnvoll sein.
- Die technischen Elemente sind in einem eigenen Bundle zusammengefasst. Dadurch sind alle Infrastruktur-Elemente an einer Stelle konzentriert.

Um die Beziehungen zwischen diesen Projekten zu verstehen, muss man zwei unterschiedliche Ebenen betrachten: Zum einen werden Packages von einem Bundle exportiert und in einem anderen Bundle importiert. Zum anderen werden von bestimmten Bundles OSGi-Services zur Verfügung gestellt und von anderen genutzt.

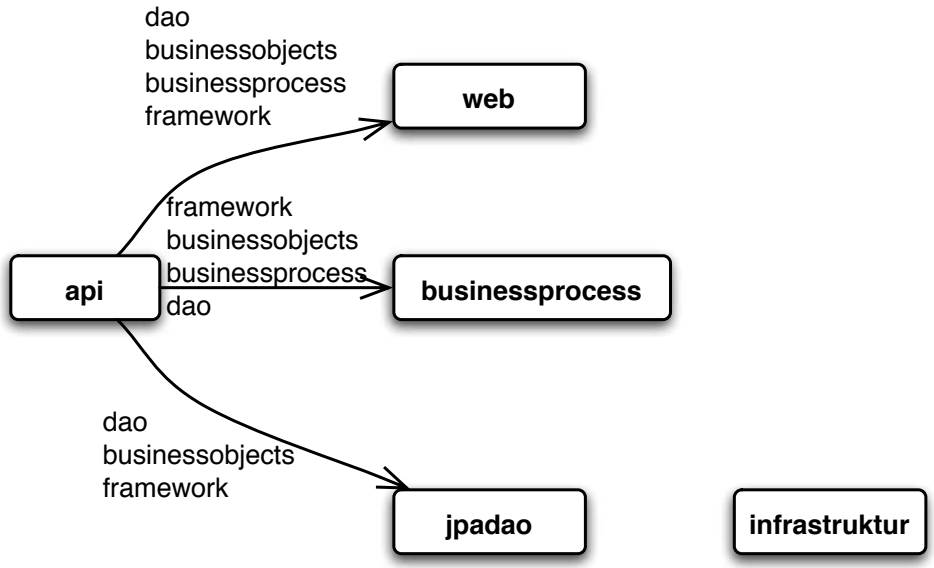

Abb. 6-1 Export und Import von Packages in der OSGi-Beispielanwendung

Import und Export von Packages im Beispiel

Abbildung 6-1 zeigt die Exporte und Importe von Packages. Dabei werden nur die Packages aus der Anwendung betrachtet. Es wird also beispielsweise nicht gezeigt, welche Bundles welche Packages aus dem Spring-Framework oder einer anderen Infrastruktur-Bibliothek nutzen. Wie man sieht, bietet nur das API-Bundle Packages an, die von anderen Bundles importiert werden. Das bedeutet, dass bezüglich der Package-Importe und -Exporte alle anderen Bundles nur von dem API-Bundle abhängen und nicht untereinander. Man kann also wie erwähnt ohne Weiteres eines der Bundles mit den Implementierungen deinstallieren, ohne dass plötzlich Packages mit Interfaces nicht mehr zur Verfügung stehen. Das ist die Voraussetzung, um eine Implementierung gegen eine andere auszutauschen. Also ist die Aufteilung nicht nur architektonisch sinnvoll, sondern bietet auch technische Vorteile.

Import und Export von OSGi-Services im Beispiel

Abbildung 6-2 zeigt den Export und Import von OSGi-Services in der Beispielanwendung. Vereinfacht wird hier ein direkter Export und Import gezeigt; in der Realität werden die Service in die OSGi-Service-Registry exportiert und von dort importiert.

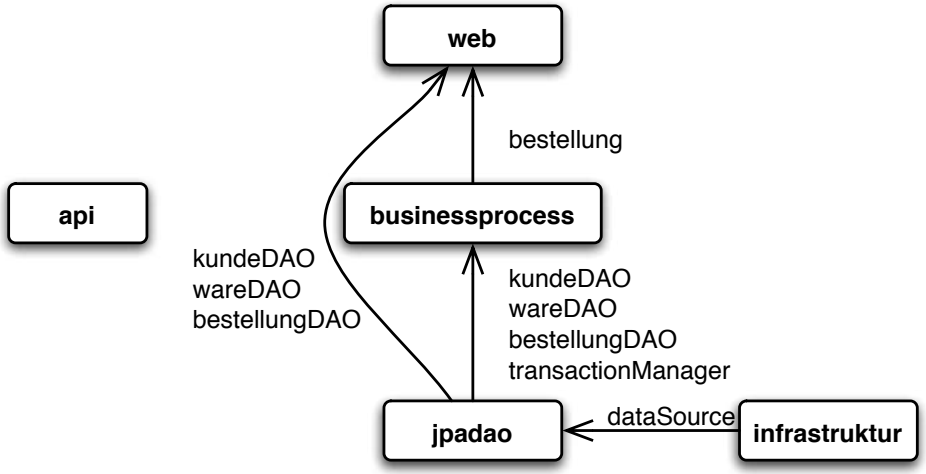

Abb. 6-2 *Export und Import von OSGi-Services in der OSGi-Beispielanwendung*

Wie man sieht, wird die `DataSource` von der Infrastruktur an die DAOs exportiert. Die DAOs werden an die Business-Prozesse exportiert und auch an die Weboberfläche. Die direkte Nutzung von DAOs in der Weboberfläche wird von einigen Architekten abgelehnt, die stattdessen die Einführung eines Business-Prozesses verlangen, der allerdings lediglich direkt an den DAO delegiert, da es keine zusätzliche Logik gibt. Um diese Durchreicher zu eliminieren, wurden die DAOs hier direkt an die Webschicht exportiert. Solche Architekturentscheidungen werden durch die OSGi-Services offensichtlich. Bei dem vorliegenden Beispiel fällt auch auf, dass der `transactionManager` von den DAOs exportiert wird – eigentlich gehört er eher in die Infrastruktur. Doch der konkrete Typ des `transactionManager` – im Beispiel ein `JpaTransactionManager` – hängt von der Persistenztechnologie ab, so dass man ihn nur mit den anderen JPA-Elementen in der DAO-Schicht unterbringen kann. Auch hier wird also eine Architekturentscheidung offensichtlich.

Durch dieses Vorgehen wird die Anwendung sinnvoll aufgeteilt, was bei großen Anwendungen die Strukturierung erleichtert und es einfacher macht, die Anwendung auch später noch zu verstehen und damit besser zu warten. Dies ist ein Hauptvorteil von OSGi: Anwendungen werden stärker modularisiert und sind dadurch einfacher zu verstehen. Diese Modularisierung wird zur Laufzeit erzwungen, so dass sie beachtet werden muss. Architekturverstöße werden so weniger wahrscheinlich. Außerdem kann man die Module zur Laufzeit austauschen, was das Einspielen neuer Versionen der Module erlaubt, ohne dass die gesamte Anwendung ausgetauscht werden müsste.

Spring-Konfiguration bei OSGi

Ein Wort noch zu den Spring-Konfigurationen im dm Server: Als Best Practice hat es sich etabliert, die Spring-Konfiguration für den Export und Import von OSGi-Services in eine Datei `osgi-context.xml` abzulegen und die Konfiguration der Spring-Beans selber in eine Datei `module-context.xml`. Neben einer sinnvollen Strukturierung ergibt sich dadurch noch ein weiterer Vorteil: Wenn man die Namen der Spring-Beans geschickt wählt, kann man dieselbe Codebasis auf dem dm Server und einer anderen, nicht OSGi-konformen Umgebung wie einem Java-EE-Applikationsserver laufen lassen.

Nehmen wir an, dass im DAO-Bundle in der Datei `module-context.xml` eine Spring-Bean mit dem Namen `wareDAO` angelegt wird. Diese wird in der Datei `osgi-context.xml` im selben Bundle als OSGi-Service exportiert. Im Business-Prozess-Bundle wird der Service in der Datei `osgi-context.xml` unter dem Namen `wareDAO` wieder importiert und dann in `module-context.xml` verwendet. Wenn man nun diese Bundles als normale JAR-Files verwendet

und einen `ApplicationContext` erzeugt, der seine Konfiguration aus `classpath*:/META-INF/spring/module-context.xml` liest, funktioniert das System immer noch. Aus dem DAO-JAR wird die Definition der Bean `wareDao` ausgelesen. Diese wird dann im Business-Prozess-JAR verwendet. Es wird nun im Business-Prozess-JAR direkt die Spring-Bean `wareDAO` aus dem DAO-Projekt verwendet, ohne dass der Umweg über den OSGi-Service genutzt wird. OSGi wird sozusagen »kurzgeschlossen«. Dadurch kann man dieselbe Anwendung im dm Server oder in einem Java-EE-Applikationsserver laufen lassen. Allerdings verzichtet man in dem Java-EE-Applikationsserver auf die Vorteile der OSGi-Umgebung wie dynamische Services und Versionsmanagement der Packages.

7 Noch ein Beispiel: Pet Clinic

Eine weitere Beispielanwendung ist Pet Clinic. Man kann es zusammen mit dem dm Server herunterladen (siehe Kapitel 8). Pet Clinic ist vielen Spring-Anwendern bekannt, denn es ist schon länger eine Beispielanwendung für das Spring-Framework. Die mitgelieferte `readme.txt`-Datei beschreibt detailliert, wie man die Anwendung bauen kann und installieren kann.

Die Anwendung ist interessant, weil sie einige Möglichkeiten des dm Servers praktisch zeigt. Die Anwendung ist in verschiedenen Bereichen flexibel:

- Man kann als Datenbank HSQLDB oder MySQL nutzen.
- Die Persistenz kann mit der JPA-Implementierung Eclipse Link, Hibernate oder JDBC implementiert werden.
- Die Anwendung ist daher in die folgenden Bundles aufgeteilt:
- `org.springframework.petclinic.domain` oder kürzer `domain` enthält die Domänen-Objekte, die ein Haustier, einen Besitzer oder einen Arzt darstellen können.
- `repository` enthält die Schnittstellen der Persistenzschicht.
- `infrastructure.mysql` und `infrastructure.hsql` bieten jeweils die Spring Beans an, die für den Zugriff auf die Datenbank – HSQLDB oder MySQL – notwendig ist.

- `jdbc`, `hibernate` und `jpa` enthalten jeweils die Implementierung der Repositories mit einer Persistenztechnologie.
- Die eigentliche JPA-Implementierung wird im Bundle `eclipselink` festgelegt.
- Schließlich gibt es ein Bundle `web`, das die Webschnittstelle enthält

JPA: Details Interessant an dieser Aufteilung ist, dass die Implementierung aus dem `jpa`-Bundle keine Hinweise darauf enthält, welche JPA-Implementierung benutzt wird. Sie importiert daher auch keine spezifische JPA-Implementierung. Die eigentliche JPA-Implementierung wird im `eclipselink`-Bundle festgelegt. Dies ist möglich, da Spring das `JpaVendorAdapter`-Interface mit Implementierungen wie dem `EclipseLinkJpaVendorAdapter` zur Verfügung stellt. Diese Klasse kann dann konfiguriert werden, um beispielsweise festzulegen, welche Datenbank genutzt wird. In dem vorliegenden Beispiel exportiert das `eclipselink`-Bundle einen OSGi-Service, der `JpaVendorAdapter` implementiert. Dieser wird dann vom `jpa`-Bundle importiert, so dass die Konfiguration im `eclipselink`-Bundle geschehen kann.

Zum anderen müssen natürlich die Klassen von Eclipse Link in der gesamten Anwendung zur Verfügung stehen. Dazu wird folgenender Import im eclipselink Bundle genutzt:

```
Import-Bundle: com.springsource.org.eclipse.
persistence;version=
 "[1.0.0,1.1.0)";import-scope:=application
```

Schließlich muss die JPA-Implementierung noch dazu in der Lage sein, die Domänen-Objekte zu ändern, um bei der Navigation zwischen Domänen-Objekten eingreifen zu können. Dies ist notwendig, da man in einigen Situationen Objekte aus der Datenbank nachladen muss. Dazu ist in der Spring-Konfiguration des `jpa`-Bundle lediglich durch ein `context:load-time-weaver`-Element Load Time Weaving aktiviert worden. Alle Spring-Beans, die Byte-Code ändern wollen, bekommen dann das dafür notwendige Objekt injiziert und können dadurch den Byte-Code der gesamten Anwendung modifizieren.

Anwendungen und PARs

Aus den oben genannten Bundles kann man anschließend eine Anwendung zusammenstellen. Davon gibt es bei diesem Beispiel drei, eine pro Persistenzlösung. Diese Anwendungen können dann als PAR in den dm Server deployt werden. Außerdem wird damit festgelegt, welche Klassen von der JPA-Implementierung bzw. von Hibernate modifiziert werden können.

Package-Export und -Import

Abbildung 7-1 zeigt die Beziehungen bei dem Import und Export der Packages in der Pet-Clinic-Anwendung. Das Package `domain` wird vom `domain`-Bundle exportiert und vom `repository`-Bundle importiert, das im `repository`-Package die Schnittstellen für die Repositories exportiert. Zusammen bilden die beiden Packages `repository` und `domain` die Schnittstelle, die vom `web`-Bundle importiert wird, um sie zu nutzen, und vom `jpa`-Bundle importiert wird, um sie zu implementieren. Die Anwendung enthält keine komplexe Geschäftslogik, daher gibt es keinen Service-Layer wie man vielleicht erwarten würde.

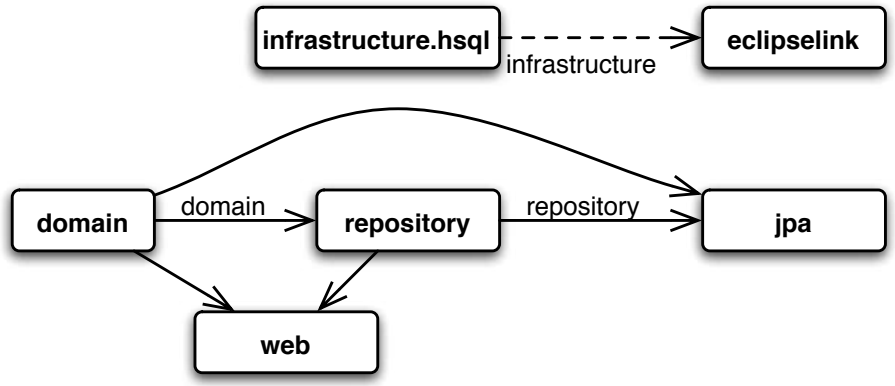

Abb. 7-1 Package-Export und -Import bei der Pet-Clinic-Anwendung

In Abbildung 7-2 ist der Export und Import von OSGi-Services dargestellt. Das `jpa`-Bundle importiert die `dataSource` aus der Infrastruktur und den `jpaVendorAdapter` vom `eclipselink`-Bundle. Selber exportiert es die `osgiClinic` an das `web`-Bundle. Das `repository`-Bundle und das `domain`-Bundle exportiert nur Packages, keine Services.

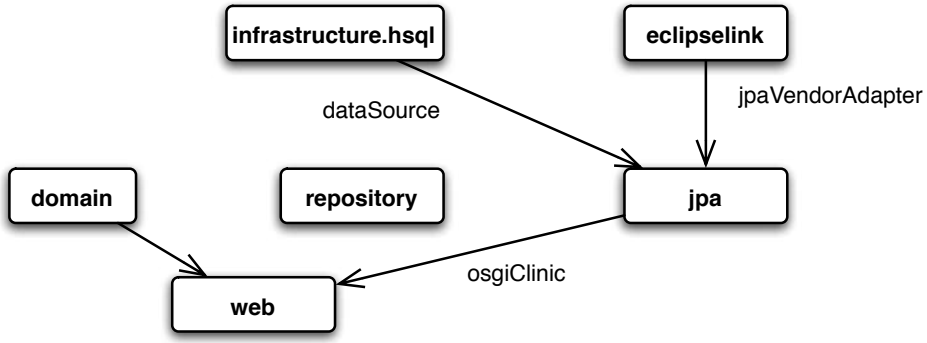

Abb. 7-2 OSGi-Service-Export und -Import in der Pet-Clinic-Anwendung

8 Installation und Betrieb

Der dm Server steht als Download unter http://www.springsource.org/dmserver zur Verfügung. Er ist eine ZIP-Datei, die man lediglich entpacken muss.

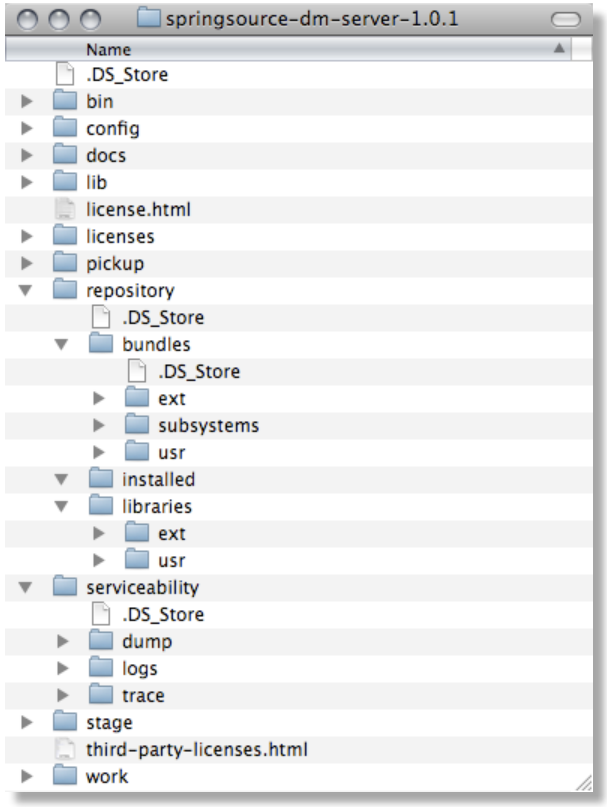

Abb. 8-1 Verzeichnisse in einer SpringSource-dm-Server-Installation

Verzeichnisstruktur Innerhalb der Distribution gibt es folgende Verzeichnisse:

- `bin` enthält Skripte zum Starten und Stoppen des Servers.
- In `config` sind verschiedene Konfigurationsdateien abgelegt zum Beispiel für den Servlet-Container. Die Einstellungsoptionen entsprechen dem Apache Tomcat, so dass man dieselben Tuning-Optionen wie bei einem Tomcat hat. Außerdem wird hier auch die Verzeichnisstruktur definiert, die man also auch an eigene Anforderungen anpassen kann.
- Das `docs`-Verzeichnis enthält zum einen den *Programmer Guide* für Entwickler und den *User Guide* für Administratoren.
- Im `lib`-Verzeichnis sind die Bestandteile des Servers selbst abgelegt.
- Das `license`-Verzeichnis enthält die Lizenzen der verschiedenen Bestandteile des dm Servers.
- Das `repository`-Verzeichnis enthält einzelne OSGi-Bundles, die als Teil der Infrastruktur von den Anwendungen auf dem dm Server genutzt werden können. Das `installed`-Unterverzeichnis wird vom dm Server intern genutzt und sollte daher immer leer sein. Das `bundle`-Unterverzeichnis enthält alle Bundles, die der dm Server zur Laufzeit zur Verfügung stellen kann. Bundles, die vom Nutzer installiert werden, müssen im Verzeichnis `usr` untergebracht werden. Also kann man den Server einfach erweitern, indem man weitere Bundles in dieses Verzeichnis kopiert. Als Quelle solcher Bundles bietet sich das *SpringSource Enterprise Repository* an, dass unter http://springsource.com/repository zur Verfügung steht und OSGi-kompatible Versionen verschiedener Open-Source-Projekte enthält. Das SpringSource Enterprise Repository läuft übrigens auf einem dm Server. Das Unterverzeichnis `ext` enthält Bundles, die zum dm Server gehören, und `subsystems` ist zur internen Nutzung durch den dm Server. Schließlich enthält das `libraries`-Verzeichnis Bibliotheken, die ja aus mehreren Bundles bestehen können. Dabei unterscheidet man die vom dm Server bereitgestellten Libraries aus dem `ext`-Verzeichnis und die vom Benutzer im `usr`-Verzeichnis bereitgestellten. Diese Libraries sind übrigens recht einfach aufgebaute Dateien, die lediglich definieren, welche Bundles zu einer Library gehören. Übrigens wird in der Dokumentation des dm Servers auch gezeigt, wie man statt dieses Repositorys zum Beispiel ein Maven-Repository nutzen kann.
- Um neue Anwendungen, Bundles oder WAR-Dateien zu deployen, kann man sie einfach in das `pickup`-Verzeichnis kopieren. Diese Dateien sind Teile von Anwendungen, während die im `repository` abgelegten Dateien zur Infrastruktur gehören.
- Das `serviceability`-Verzeichnis enthält Dateien, die bei der Analyse von Anwendungen helfen. Dazu gehören zunächst die im Unterverzeichnis `dump` untergebrachten Dumps. Diese Dateien werden automatisch erstellt, wenn ein kritischer Fehler wie zum Beispiel ein Deadlock vorkommt, und enthalten alle wesentlichen Informationen über den Zustand der JVM zu diesem Zeitpunkt, also zum Beispiel über den Zustand der Threads und der geladenen Bundles. Das `log`-Verzeichnis enthält die Logausgaben des Servers selbst und zeigt somit, welche Ereignisse bei der Ausführung des Servers vorgekommen sind. Schließlich enthält das `trace`-Verzeichnis die Logausgaben der deployten Bundles. Dabei werden die Ausgaben der verschiedenen Java-Logging-APIs wie log4j oder Commons Logging, aber auch die Ausgaben an `System.out` oder `System.err` in diese Datei umgeleitet. Was genau in diesen Logdateien landet, kann man durch einen Eintrag in das `META-INF/MANIFEST.MF` des Bundles definieren. Dort kann man eine Zeile wie `Application-TraceLevels: *=info,com.*=verbose` unterbringen. Dadurch werden für alle Klassen Logs mit Ausgaben erstellt, die mindestens die Priorität »Info« haben. Alle Klassen, die aus einem Package kommen, dessen

Name mit com beginnt, werden ausführlich (»verbose«) geloggt. Ähnliche Einstellungen, die dann aber für den gesamten Server gültig sind, kann man auch in der Datei `server.config` im `config`-Verzeichnis vornehmen.

- Die `stage`- und `work`-Verzeichnisse sind zur internen Nutzung durch den dm Server.

Admin-Anwendung

Wem das Arbeiten auf dem Dateisystem nicht entgegenkommt, kann auch die Admin-Anwendung nutzen, die unter `http://localhost:8080/admin/` zur Verfügung steht. Sie ermöglicht zum Beispiel das Deployen von Anwendungen.

Eclipse-Plugin

Eine weitere Möglichkeit zum Umgang mit dem dm Server ist das Eclipse-Plugin, das man zusammen mit dem Server herunterladen kann. Mit dem Plugin kann man den dm Server unter der Kontrolle von Eclipse starten und dadurch die Anwendungen auch sehr einfach debuggen. Außerdem kann man Bundles durch Drag-and-drop auf dem Server installieren. Das Repository des dm Servers kann man ebenfalls durch das Eclipse-Plugin verwalten. Es ermöglicht, Bundles aus dem SpringSource Enterprise Bundle Repository in dem dm Server zu installieren. Außerdem kann man auch die Abhängigkeiten zwischen den Bundles im Server visualisieren. Ein Beispiel zeigt Abbildung 8-2.

Außerdem steht eine Konsole für den Server unter dem Port 2401 zur Verfügung. Man kann also durch ein `telnet localhost 2401` eine Verbindung zum dm Server einrichten. Die folgenden sind die wichtigsten Kommandos:

- `help` listet die möglichen Kommandos mit einer kurzen Erklärung auf.
- `ss` (short status) gibt Informationen über die installierten Bundles und ihren Status aus.
- `status` gibt den vollständigen Zustand inklusive aller Services und Bundles aus.
- Mit `start` können Bundles gestartet werden, und mit `stop` werden sie gestoppt. Das Bundle wird durch eine Nummer identifiziert, die man zum Beispiel durch `ss` ermitteln kann.
- Mit `install` kann man ein neues Bundle installieren. Dabei kann man die URL angeben, von der das Bundle installiert werden soll. Mit `uninstall` kann es dann wieder entfernt werden.

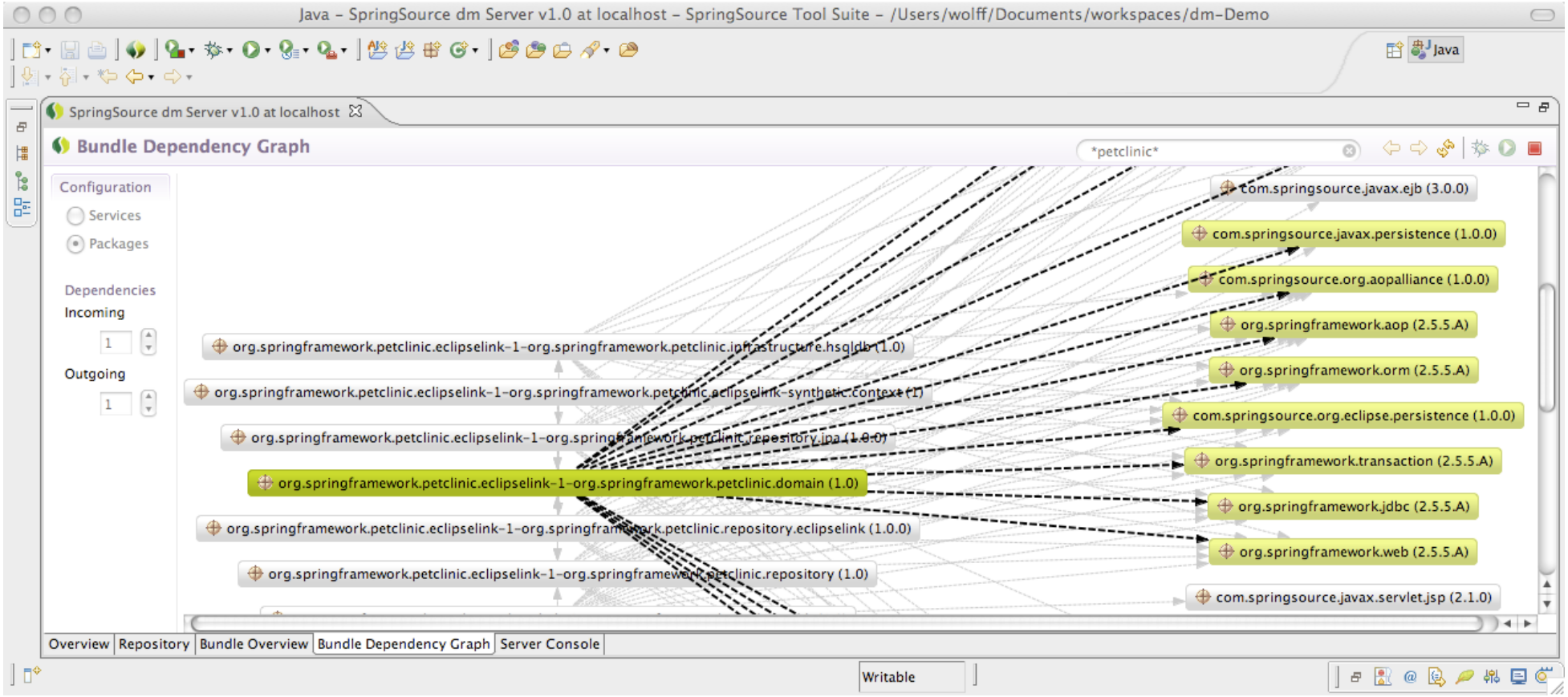

Abb. 8-2 *Bundle-Abhängigkeiten im dm Server Eclipse Plugin*

Bibliografie

[Equinox] http://www.eclipse.org/equinox/
Diese OSGi-Implementierung stellt die Basis für die Eclipse-Plattform dar. Sie ist aber in ihrem Einsatz nicht auf die Entwicklung von Entwicklungswerkzeugen beschränkt.

[Eva03] Eric Evans:
Domain-Driven Design, Addison-Wesley Longman, 2003
In diesem Buch beschreibt Eric Evans einige wesentliche Techniken für die Entwicklung von objektorientierten Systemen. Dabei fokussiert er vor allem darauf, wie man fachliche Konzepte sinnvoll im Code implementieren kann.

[Fow02] Martin Fowler et al:
Patterns of Enterprise Application Architecture, Addison-Wesley, 2002
Dieses Buch enthält eine sehr lesenswerte Sammlung von Patterns für die Entwicklung von Enterprise-Systemen. Dazu zählen neben Persistenzansätzen auch der Umgang mit Zustandsinformationen. Dieses Buch sollte jeder gelesen haben, der große Enterprise-Systeme entwickeln will.

[Knopflerfish] http://www.knopflerfish.org/
Knopflerfish ist eine Open-Source-OSGi-Plattform.

[OSGi] http://www.osgi.org/
Der OSGi-Standard definiert ein Komponentenmodell, das auf Java basiert. Ursprünglich war es eher für den Embedded-Bereich gedacht, aber es löst allgemeine Probleme z. B. im Bereich Class-Loading, so dass es mittlerweile die Basis verschiedener Projekte wie z. B. der Eclipse-Entwicklungsumgebung ist.

[Spring] http://spring-buch.de/, E. Wolff:
Spring 2 – Framework für die Java-Entwicklung, dpunkt.verlag, 2008
Dieses Buch bietet eine umfangreiche Einführung in das Spring-Framework und darauf basierende Technologien wie Spring Web Services oder Spring Web Flow. Eine Neuauflage zu Spring 3 wird im Laufe des Jahres 2009 erscheinen.

[SpringOSGi] http://www.springsource.org/osgi/
Dieses Projekt implementiert eine Integration von OSGi und Spring, so dass man das feingranulare Spring-Komponentenmodell mit dem eher grobgranularen OSGi-Komponentenmodell zusammen verwenden kann.

[WHKL08] G. Wütherich, N. Hartmann, B. Kolb, M. Lübken:
Die OSGi Service Platform, dpunkt.verlag, 2008
http://www.dpunkt.de/buecher/2635.html
Dieses Buch stellt OSGi sehr ausführlich dar und eignet sich daher sehr gut als Lektüre zu diesem Thema. Dabei werden neben dem OSGi Framework auch verschiedene Standard-OSGi-Services vorgestellt.

Lektorat: René Schönfeldt
Herstellung: Nadine Berthel

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-89864-955-1

1. Auflage 2009

Ringstraße 19
69115 Heidelberg

5 4 3 2 1