Wolfgang KOPP

Modulare Softwarearchitektur in JAVA mit Fokus auf JAVA Enterprise Edition

Bachelor Arbeit

Institut für Genomik und Bioinformatik
Technische Universität Graz
Petersgasse 14, 8010 Graz
Vorstand: Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Betreuer:
Dipl.-Ing. Dr.techn. Gernot Stocker

Begutachter:
Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Graz, Mai 2009

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

Zusammenfassung

Für die Entwicklung modularer Software ist es erforderlich, eine monolithische Anwendung in kleine Teile zu zerlegen, die voneinander unabhängig sind und über Schnittstellen interagieren. JAVA unterstützt die theoretischen Anforderungen an die Entwicklung modularer Software nicht, jedoch bietet das standardisierte OSGi Framework die Möglichkeit, modulare serviceorientierte JAVA-Anwendungen zu entwickeln. Zudem wird OSGi dynamischen Anforderungen sowohl auf Klassen- als auch auf Objektebene gerecht, ist allerdings nicht für verteilte oder serverseitige Anwendungen konzipiert.

Im Gegensatz zum hierarchischen Class Loading System in JAVA-SE- oder JAVA-EE-Anwendungen basiert OSGi auf einem graphenähnlichen Class Loading System, das den Vorteil bietet, Softwaremodule unabhängig voneinander und dynamisch verwalten zu können.

Obwohl die Integration von OSGi in JEE Anwendungen zu deren Verbesserung beitragen würde, wird die OSGi-Technologie noch nicht in der JEE Spezifikation berücksichtigt. Alternativ zu JEE unterstützt der SpringSource dm Application Server eine Kombination aus serverseitiger Anwendung mit OSGi. Die daraus resultierende Anwendung ist versionierbar, modularisierbar und zur Laufzeit veränderbar.

Auch das Newton Framework bietet eine hohe Flexibilität im Bezug auf veränderliche Anforderungen, die auf die integrierte OSGi Technologie zurückzuführen ist. Anders als der dm Server unterstützt Newton allerdings das Anwendungsfeld der verteilten Systeme. Newton-Anwendungen sind als komponentenbasierend und serviceorientiert zu charakterisieren, die sowohl manuell als auch automatisch verwaltet werden können.

An Produkten wie dem Newton Framework oder dem dm Server zeichnet sich der Trend für zukünftige Entwicklungen ab: Sie bewegen sich weg von monolithischen und statischen, hin zu modularen und dynamischen Anwendungen.

Schlüsselwörter

JAVA, OSGi, Spring, J2EE, Verteilte Systeme;

Inhaltsverzeichnis

1 Einleitung

Modularisierung

Die Entwicklung modularer Software war bereits seit Beginn der Siebzigerjahre ein wichtiges Forschungsthema (siehe [1]). Die wesentlichen Anliegen dabei sind, die Softwarekomplexität zu reduzieren, die Entwicklungsprozesse der verschiedenen Module voneinander zu entkoppeln, sowie die Software wartbar und wiederverwendbar zu machen. Daraus folgen vier Eigenschaften, die für modulare Software charakteristisch sind:

- *Explizite Struktur*: Damit ist die Aufteilung eines großen Software-Komplexes in mehrere, logisch voneinander unabhängige Module gemeint, denen jeweils ein bestimmtes Anliegen zugeteilt wird. Grundsätzlich geht man dabei so vor, dass man Programmelemente, die eng miteinander interagieren, in eine Gruppe zusammenfasst (modularisiert). Dieses Prinzip ist auch als high-cohesion bekannt. Außerdem erreicht man damit, dass Module untereinander einen sehr geringen Kopplungsgrad aufweisen, was als loosely coupled bezeichnet wird. Module dürfen untereinander nur über definierte Schnittstellen kommunizieren. Wenn eine Schnittstelle von einem Client-Modul verwendet wird, muss diese explizit importiert werden. Da das Modul nur über Schnittstellen nach außen geöffnet ist, ist die konkrete Implementierung, wie schon von der objektorientierten Programmierung her bekannt, nach außen nicht sichtbar.
- *Separate Entwicklung*: Da Module nur über Schnittstellen interagieren, können sie unabhängig voneinander implementiert, übersetzt und mittels eines Testframeworks separat getestet werden. Derartig modularisierte Programmierung erlaubt eine parallele Entwicklung und führt damit zur Einsparung von Übersetzungszeit und zur Wiederverwendbarkeit von Modulen.
- *Informationen verbergen*: Eine grundlegende Fähigkeit modularer Softwarearchitekturen ist es, gegen Schnittstellen programmieren zu können, um Implementationsdetails verborgen zu halten. Der Vorteil, der sich dabei ergibt, ist, dass die Implementation eines Moduls leicht geändert werden kann, ohne dabei die Schnittstelle ändern zu müssen.
- *Wiederverwendbarkeit*: Aus den bisher erwähnten Eigenschaften emergiert die Möglichkeit, Module wiederzuverwenden. In Bezug auf die Wiederverwendbarkeit von Modulen sind folgende Teilaufgaben notwendig: 1. Suchen von passenden Modulen, 2. Anpassen der Module an den jeweiligen Kontext, 3. Zusammenstellen mehrerer wiederverwendeter oder neuer Module zu einer Komposition, die das gesamte Softwaresystem ergibt (siehe [3] und [15]).

JAR

In JAVA ist die Standard Deployment Einheit die JAR-Datei. Ein JAR erfüllt eine ähnliche Aufgabe wie ein ZIP-Archiv. Es können damit Klassen und Ressourcen logisch gruppiert und verpackt werden. Diese Pakete werden der Laufzeitumgebung z.B. über das Setzen des Klassenpfades zur Verfügung gestellt.

JARs sind aber nicht in der Lage, Softwaremodule, wie sie zuvor spezifiziert wurden, zu bilden. Dies folgt aus der Tatsache, dass es keine Mechanismen für JARs gibt, um Informationen zu verbergen. In JARs werden beispielsweise keinerlei Meta-Daten über das Exporten und Importen von Schnittstellen abgelegt. Des Weiteren sind alle Klassen und Ressourcen, die in einem JAR verpackt wurden, global zugänglich [4].

Damit trotzdem modulare Software in JAVA SE erstellt werden kann, bestünde nun die Möglichkeit, Konventionen einzuführen, in denen die obigen Anforderungen berücksichtigt werden, z.B. könnten Package-Bezeichnungen auf Modulinterna und -externa hinweisen. Ein Client dürfte dann nur solche Klassen in Packages verwenden, die für eine externe Verwendung spezifiziert sind.

Dieses Konzept ist allerdings kein wirklicher Schutz vor einer missbräuchlichen Verwendung modulinterner Klassen und Ressourcen durch den Client.

Flexibilität

Ein weiteres wichtiges Anliegen der modernen Softwareentwicklung ist eine erhöhte Flexibilität der Software. Eine Anwendung sollte den Anforderungen der realen Welt so gut wie möglich entsprechen, aber auch den Faktor der Änderung von Anforderungen berücksichtigen. Da die reale (d.h. für die Datenverarbeitung abzubildende) Welt häufig Änderungen unterworfen ist, müssen die Softwaremodelle diesen entsprechend angepasst werden. Bei den in weiterer Folge evaluierten Technologien soll deshalb auch die Flexibilität als Bewertungskriterium ins Auge gefasst werden.

Im Bereich von J2EE Anwendungen ist es beispielsweise möglich, EJB Komponenten auf verschiedene Container zu verteilen. Die Komponenten werden so verteilt, dass Komponenten mit einer hohen Affinität zueinander in einem Container installiert werden. Komponenten mit einer schwächeren Kopplung werden auf externe Container verteilt, die über ein Kommunikationsnetz miteinander verknüpft sind (z.B. RMI oder Webservices). Bei Änderungen in den Anforderungen könnten sich etwa auch die Affinitäten ändern, wobei hier auf einfache Weise die Komponenten von einem Container zum nächsten überführt werden können.

Für detaillierte Ausführungen zu verteilten Systemen und ihrer Charakteristika siehe [6].

2 Methoden

2.1 OSGi

2.1.1 Überblick

OSGi ist ein standardisiertes Framework für JAVA Programmierer, welches für modulare und dynamisch veränderbare Software konzipiert wurde. Mit OSGi ist es möglich, Softwaremodule zu entwickeln, die während der Laufzeit der Anwendung installiert, geändert bzw. deinstalliert werden können. Die OSGi Spezifikation und deren Weiterentwicklung liegt in der Verantwortung der OSGi Alliance. Die OSGi Alliance ist eine nicht-profitorientierte Gesellschaft, die 1999 gegründet wurde. Sie setzt sich aus Kompetenzen verschiedener Marktsegmente, wie Unternehmens-, Mobil- oder eingebetteten Anwendungen, zusammen. Dadurch fließen Aspekte verschiedener Märkte in die Spezifikation ein.

Das OSGi Framework kann vereinfacht als Schichtmodell dargestellt werden (siehe Abbildung 1).

Aufgesetzt auf JAVA SE beschreibt die Schicht „Execution Environment" eine Implementation der OSGi Plattform. Gängige OSGi Implementationen sind Eclipse Equinox, Apache Felix oder Knopflerfish.

Die Modulschicht beinhaltet die Beschreibung zur Assemblierung von Klassen und Ressourcen zu Modulen, wie sie in der Einleitung definiert wurden. Dazu müssen Meta-Informationen über das Modul abgelegt werden. Weiters wird in dieser Schicht beschrieben, wie die explizite Angabe von benötigten und bereitgestellten Schnittstellen in OSGi realisiert wird. Die genaue Beschreibung der Anatomie eines Modules in OSGi und deren Kommunikationsschnittstellen folgt in den Abschnitten 2.1.2 und 2.1.3.

In der Life-Cycle-Schicht wird beschrieben, wie Module auf der OSGi Service Platform, zur Laufzeit des Programms, installiert, geändert und wieder deinstalliert werden. Diese Fähigkeit von OSGi unterstützt ein hoch dynamisches Verhalten der Anwendung. In Abschnitt 2.1.4 wird darauf näher eingegangen.

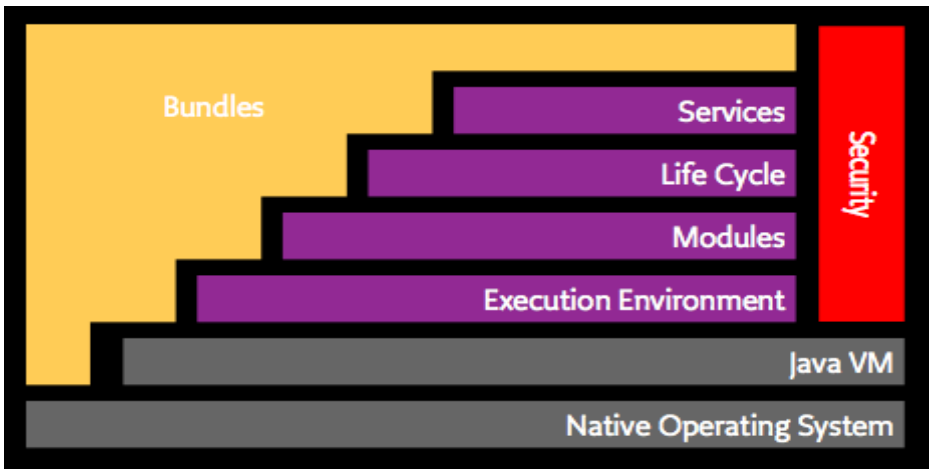

Abbildung 1: OSGi Schichtmodell (Quelle: [11])

Die Service-Schicht behandelt die Ebene der JAVA Objekte und zielt auf die Entkopplung der Anliegen auf Objektebene ab. OSGi ist für serviceorientierte Anwendungen ausgelegt. Typischerweise werden OSGi Services als POJOs (das sind einfache JAVA Klassen, die von keiner Framework-Schnittstelle abhängig sind) implementiert und dann über ein JAVA-`interface` an einer zentralen Stelle zum Konsum zur Verfügung gestellt. Diese zentrale Stelle wird die OSGi Service Registry genannt. Das Konzept der OSGi Service Registry ähnelt stark dem von Namens- und Verzeichnisdiensten (Siehe [6] und [8]). Auch die Service-Schicht ist, wie die Modul-Schicht, für hohe dynamische Anforderungen der Anwendung ausgelegt. OSGi Services werden in Abschnitt 2.1.5 noch genauer behandelt.

Die Security-Schicht durchzieht die bisher besprochenen Schichten gemäß Abbildung 1. Einerseits wird damit geregelt, welche Module auf diverse externe Klassen und Ressourcen Zugriff haben, andererseits kann z.B. auch der Zugriff auf OSGi Services mit Zugriffsrechten geregelt werden. Die Sicherheitsschicht wird in Abschnitt 2.1.8 beschrieben.

Für weitere detaillierte Beschreibungen siehe [5-11].

2.1.2 OSGi Bundles

In OSGi werden abgeschlossene Module als Bundles bezeichnet, wobei Modulinterna nach außen verborgen und Schnittstellen gezielt nach außen bereitgestellt werden. Die Konzepte der OSGi Bundles sind in der Modulschicht (Abbildung 1) angesiedelt.

Typischerweise ist ein Bundle ein JAR-File, das neben den Klassen und Ressourcen noch eine Beschreibung über sich selbst beinhaltet. Diese Meta-Informationen werden in einem „MANIFEST.MF" abgelegt.

Ein typisches MANIFEST.MF könnte zum Beispiel folgende Daten enthalten:

```
Bundle-SymbolicName: HelloWorldBundle
Bundle-ManifestVersion: 2
Bundle-Name: Hello World Bundle
Bundle-Version: 1.0.1
Bundle-Activator: HelloWorldActivator
Import-Package: org.osgi.framework;version="1.4.0"
```

Der `Bundle-SymbolicName` ergibt zusammen mit der `Bundle-Version` eine eindeutige Identifikation des Bundles im Framework. Dabei ist der `Bundle-SymbolicName` zwingend anzugeben, im Gegensatz zu allen übrigen Manifest Headern, die optional genutzt werden können. `Bundle-Activator` gibt die Activator Klasse des Bundles an, welche den

Einsprungspunkt des Frameworks in das Bundle beim Starten und Stoppen darstellt. `Import-Package` ist nötig, um Schnittstellen eines anderen Bundles zu nutzen. Das Framework löst zur Deployment-Zeit alle nötigen Abhängigkeiten des zu ladenden Bundles zu anderen Bundles auf. Eine genauere Ausführung zum expliziten Importen bzw. Exporten von Schnittstellen folgt in Abschnitt 2.1.3.

Ein Activator, wie er im obigen MANIFEST.MF-Ausschnitt referenziert wurde, soll das Beispiel vervollständigen:

```
import org.osgi.framework.*;

public class HelloWorldActivator implements BundleActivator {
  public void start(BundleContext context) {
    System.out.println("Hello World!");
  }

  public void stop(BundleContext context) {
    System.out.println("Goodbye World!");
  }
}
```

Wesentlich dabei ist, dass die Klasse `HelloWorldActivator` die Schnittstelle `BundleActivator` implementiert:

```
package org.osgi.framework;

public interface BundleActivator {
  public void start(BundleContext context) throws Exception;
  public void stop(BundleContext context) throws Exception;
}
```

Die Callback-Methoden `start()` und `stop()` werden vom Framework aufgerufen, wenn das Bundle gestartet bzw. gestoppt wird.

Beide Methoden bekommen als Argument ein Objekt der Klasse BundleContext übergeben. Dieser stellt eine zentrale Schnittstelle zum OSGi Framework dar. Über den BundleContext können beispielsweise die Lebenszyklus-Zustände (siehe Abschnitt 2.1.4) anderer Bundles beobachtet oder manipuliert werden. Außerdem werden über den BundleContext Methoden für den lesenden und schreibenden Zugriff auf die OSGi Service Registry zur Verfügung gestellt (siehe Abschnitt 2.1.5 und 2.1.6).

Für genauere Informationen siehe [11].

2.1.3 Package-Abhängigkeiten

Dieser Abschnitt behandelt das explizite Bereitstellen bzw. Konsumieren von Modul-Schnittstellen, was ein wichtiger Mechanismus für modulare Software ist.

In OSGi wird dieser Mechanismus über JAVA-packages erreicht. Es werden diejenigen Packages exportiert, welche öffentliche Schnittstellen enthalten. Das OSGi Framework

stellt sicher, dass Packages, die nicht explizit exportiert wurden, nach außen nicht sichtbar sind.

Export

Der Export eines Packages soll anhand eines Beispiels gezeigt werden. Ein Bundle beinhaltet Klassen, die in zwei Packages aufgeteilt werden:

`org.foo.interfaces`: Dieses Package beinhaltet Schnittstellen (`interface` siehe [14]), die nach außen hin veröffentlicht werden. Hier könnten natürlich auch JAVA Klassen veröffentlicht werden, aber um guter Programmierpraxis zu folgen sollten in den meisten Fällen JAVA interfaces verwendet werden.

`org.foo.internal`: Dieses Package beinhaltet Klassen, die nach außen nicht sichtbar sein sollen. Darüber hinaus werden unter anderen auch die Schnittstellen des `org.foo.interfaces` Packages implementiert.

Im MANIFEST.MF hat nun folgende Erweiterung zu erfolgen:

```
Bundle-SymbolicName: A

Export-Package: org.foo.interfaces
```

Damit wird dem Framework eine öffentliche Schnittstelle zum Bundle geboten.

Import

Um die Ressourcen und Klassen aus anderen Bundles zu nutzen, müssen sie durch einen Zusatz im MANIFEST.MF explizit importiert werden. Hier gibt es grundsätzlich zwei Möglichkeiten:

```
Import-Package: org.foo.interfaces
```

oder

```
Require-Bundle: A
```

`Import-Package` dient dazu, externe Ressourcen und Klassen im Modul zu nutzen. Es wird genau dieses Package importiert und sonst nichts. Im Gegensatz dazu werden bei `Require-Bundle` alle Packages importiert, die ein Bundle veröffentlicht.

In den meisten Fällen ist vom Gebrauch von `Require-Bundle` eher abzuraten, weil man unter anderem abhängig von der Implementation wird und oft zusätzliche Packages importiert, welche vom Client nicht benötigt werden [9].

Das OSGi Framework löst gemäß der angegebenen Import- bzw. Export-Manifest Header die Package-Abhängigkeiten zur Deployment-Zeit auf. Das Framework kann die Abhängigkeiten eines Bundles nicht auflösen falls beispielsweise ein Package importiert wird, welches nicht von einem Bundle exportiert wurde. Diese Tatsache spiegelt sich in dem Lebenszyklus-Zustand eines Bundles wider, worauf in Abschnitt 2.1.4 eingegangen wird.

Versionierung von Packages und Bundles

In OSGi ist es möglich ein Bundle zu versionieren. Dazu muss lediglich ein Eintrag mit dem `Bundle-Version`-Header im MANIFEST.MF erfolgen.

Die `Bundle-Version` und der `Bundle-SymbolicName` ergeben die eindeutige Kennzeichnung des Bundles im Framework. Die Version muss drei numerische Stellen beinhalten, die für „major", „minor", „micro" stehen. Eine vierte Stelle („qualifier") kann optional angegeben werden. Wird auf die Angabe des Bundle-Version-Headers verzichtet, verwendet das Framework automatisch die Version „0.0.0".

Durch die Versionierung wird ermöglicht, dass unterschiedliche Versionen eines Bundles im Framework installiert werden können. Mit einfachen JARs ist das nicht möglich.

Unabhängig davon, ob, wie und welche Version ein Bundle bekommt, ist es andererseits auch möglich ein Package mit einer Version zu kennzeichnen. Dazu wird die Version nach

dem Export-Package-Header angegeben. Ein Beispiel für die Verwendung dieses Manifest Headers:

```
Export-Package: org.foo.interface;version="1.2.3"
```

Auch hier ist es möglich, mehrere Versionen eines Packages in das Framework zu exportieren.

Beim Importieren eines Packages oder eines Bundles kann bei Bedarf auf eine spezielle Version zurückgegriffen werden. Dabei ist es möglich, gewisse Filterkriterien und Ranking-Optionen zu nutzen. Beispiel:

```
Import-Package: org.foo.interface;version="[1.0.0,1.3.3]"

Require-Bundle: A;version="[1.0.0,2.0.0)"
```

Die Version im Import-Package bezieht sich dabei auf die Version des Packages, welche im Export-Package-Header angegeben wird. Die Version des Require-Bundle-Headers bezieht sich auf die Bundle-Version.

In den beiden Beispielen ist unter anderem der Bereich beschränkt, in dem sich die zu nutzende Version befinden soll. Die verschiedenen Klammerungen haben die Bedeutung:

„[" → größer gleich

„]" → kleiner gleich

„(" → größer

„)" → kleiner

Details dazu finden sich in Kapitel 3.5 [11].

2.1.4 Lebenszyklus eines Bundles

Ein Bundle kann zur Laufzeit der Virtual Machine jederzeit installiert, gestartet, gestoppt, geändert und deinstalliert werden. Dieses Konzept ist in die Lebenszyklus-Schicht (siehe Abbildung 1) eingeordnet. Alle möglichen Lebenszyklus Zustände eines OSGi Bundles sind in Tabelle 1 zusammengefasst.

Lebenszyklus-Zustände eines Bundles
INSTALLED
RESOLVED
STARTED
ACTIVE
STOPPED
UNINSTALLED

Tabelle 1: Lebenszyklus Zustände eines Bundles

Unmittelbar nach der Installation befindet sich das Bundle im Status INSTALLED. In diesem Zustand ist das Bundle dem Framework bekannt, aber es wurden noch keine Package-Abhängigkeiten aufgelöst. Je nach OSGi Implementation versucht das Framework dann entweder automatisch die Abhängigkeiten aufzulösen oder wartet auf einen manuellen, administrativen Eingriff.

Konnten alle Package-Abhängigkeiten erfolgreich aufgelöst werden, befindet sich das Bundle im Status RESOLVED.

Wird das Bundle gestartet, erreicht es den Zustand START, welcher so lange beibehalten wird, bis die start()-Methode des Activators erfolgreich abgearbeitet wurde. Danach befindet sich das Bundle im Zustand ACTIVE. Analog zum Starten wird beim Stoppen der Zustand STOPPING so lange beibehalten, bis die stop()-Methode abgearbeitet wurde. Danach befindet sich das Bundle wieder im Zustand RESOLVED.

Wird ein Bundle nicht mehr benötigt, kann es natürlich auch wieder deinstalliert werden.

Der Lebenszyklus eines Bundles kann entweder über einen sogenannten Management Agent, zum Beispiel die Equinox Console, oder programmiertechnisch (über den BundleContext) manipuliert werden.

Für weitere Informationen zu den Lebenszyklen eines Bundles sei auf [9] und [11] verwiesen.

2.1.5 OSGi Services

Das Konzept der OSGi Services ist in der Service Schicht (Abbildung 1) angesiedelt.

OSGi Services werden als POJOs implementiert und über Schnittstellen (JAVA `interface`) an einer zentralen Stelle zur Nutzung zur Verfügung gestellt. Diese zentrale Stelle wird als OSGi Service Registry bezeichnet. Die OSGi Service Registry erledigt eine ähnliche Funktion wie diverse Namens- und Verzeichnisdienste. OSGi Services werden mit Hilfe der OSGi Service Registry über die Modulgrenzen hinweg, d.h. global, bereitgestellt. OSGi Services können von einem Bundle publiziert und danach von einem anderen Bundle konsumiert werden. Die drei Akteure Service Publizist, Service Registry und Service Konsument sind dabei in einer Dreiecksbeziehung miteinander gekoppelt, was in Abbildung 2 verdeutlicht wird:

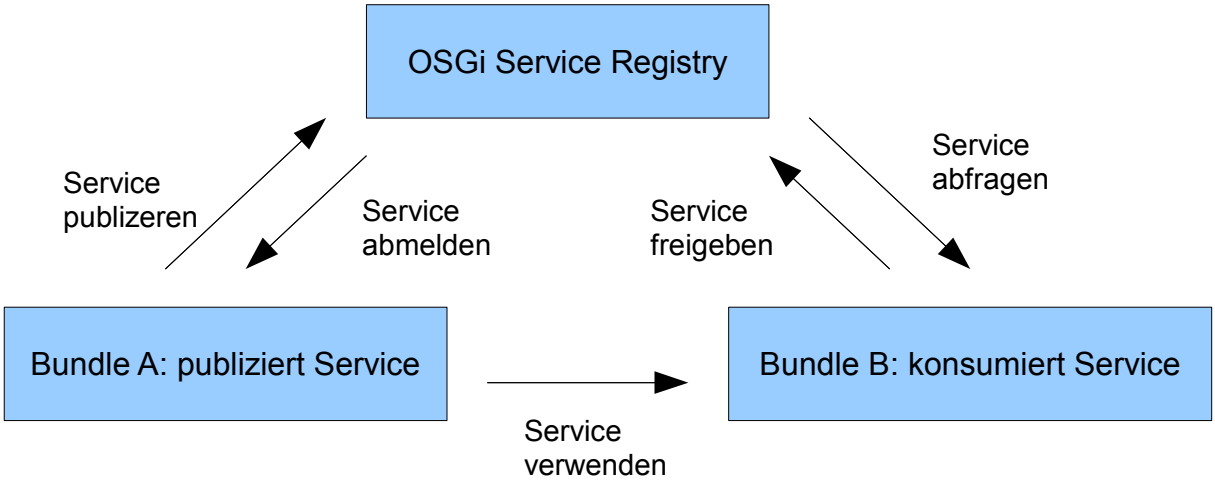

Abbildung 2: OSGi Services - Koppelung von Publizist, Service Registry und Konsument

Der `BundleContext` stellt eine zentrale Interaktionsschnittstelle zwischen Anwendung und OSGi Framework dar, daher bietet er auch alle Methoden an, um auf die OSGi Service Registry lesend und schreibend zugreifen zu können.

Service publizieren

Zur Anmeldung eines Services an die OSGi Service Registry stehen zwei Methoden zur Verfügung:

```
package org.osgi.framework;
[...]
public interface BundleContext {
  [...]
  public ServiceRegistration registerService(
      java.lang.String classname, java.lang.Object service,
      java.util.Dictionary  properties);
  public ServiceRegistration registerService(
      java.lang.String[] classnames, java.lang.Object service,
      java.util.Dictionary  properties);
  [...]
}
```

Mit der ersten Variante wird als erster Parameter (String classname) der Name der Klasse bzw. des Interfaces übergeben, der in der Regel über den Package Export nach außen bekannt gemacht wird. Als zweiter Parameter (Object service) wird eine Referenz auf das erzeugte Objekt übergeben, das die Schnittstelle implementiert. Über den dritten, optionalen Parameter (Dictionary properties) können dem Service beliebige Eigenschaften zugeschrieben werden, welche unter dem Key-Value-Prinzip abgelegt werden. Wird diese Option nicht genutzt, kann null übergeben werden.

Im Unterschied zur ersten Methode können in der zweiten mehrere Schnittstellen bzw. Klassen übergeben werden, die ein Objekt implementiert.

Zu beachten ist, dass als Rückgabewert ein Objekt der Klasse `ServiceRegistration` übergeben wird. `ServiceRegistration` ist nicht direkt das Service, welches zur Verfügung gestellt wurde, sondern eine Kapselung der Meta-Informationen für das Service. Mit dem Objekt vom Typ `ServiceRegistration` ist es möglich, im BundleContext das Service zu nutzen.

Eine typische Stelle im Code, um ein Service an die OSGi Service Registry anzumelden, ist der Activator des Bundles.

Service Factory

Die Verwendung der Service Factory ist eine weitere Methode, um Services an die OSGi Service Registry anzumelden. Diese Art, Services zu publizieren, wird immer dann benötigt, wenn jedes Client-Bundle eine eigene Instanz des Services zugewiesen bekommen soll.

Um eine Service Factory anzubieten, muss folgende Schnittstelle implementiert werden:

```
package org.osgi.framework;
public interface ServiceFactory {
  public Object getService(Bundle bundle,
            ServiceRegistration registration);
  public void unsetService(Bundle bundle,
            ServiceRegistration registration, Object service);
}
```

Da die Services gecached werden, ist es nur möglich ein Service pro Bundle zu konstruieren. Existiert für ein Bundle bereits ein Service, wird das bestehende Service weiterverwendet.

Service abmelden

Durch den Aufruf der Methode `unregister()` des Interfaces `ServiceRegistration` kann ein Service wieder aus der Service Registry entfernt werden. Wird ein Bundle gestoppt, wird diese Methode für alle Services des Bundles automatisch aufgerufen.

Um Services wieder aus der OSGi Service Registry zu entfernen, muss im Falle der ServiceFactory die Methode `unsetService()` verwendet werden.

Service abfragen

Die in der OSGi Service Registry angemeldeten Services können über die nachfolgenden Methoden aus dem BundleContext abgefragt werden:

```
public  interface BundleContext {
  […]
  public ServiceReference getServiceReference(
      String classname);
  public ServiceReference[] getServiceReference(
      String classname, String filter) throws
                                    InvalidSyntaxException;
  [...]
}
```

Beiden `getServiceReference()`-Methoden wird dazu der Klassen- bzw. Interface-Name des benötigten Services übergeben. Bei der zweiten Methode wird zusätzlich zum Klassen- bzw. Schnittstellennamen noch ein Filter-String übergeben, der es anhand der zuvor definierten Eigenschaften ermöglicht, ein passendes Service auszuwählen. Dieser Filter-String kodiert beliebige Filterkriterien mittels spezieller Operatoren. Der Filter-String muss einer speziellen Syntax entsprechen, welche auf RFC 1960, „A String Representation of LDAP Search Filters" [13] basiert.

Beide `getServiceReference()`-Methoden geben ein `ServiceReference` Objekt zurück, unter der Voraussetzung, dass das Service gerade verfügbar ist, ansonsten wird null zurückgeliefert. Abschnitt 2.1.6 behandelt die Frage, was passiert, wenn das Service nicht mehr zur Verfügung steht.

Um letztendlich das OSGi Service zu erhalten, muss folgender Beispielcode vom Client verwendet werden:

```
MyServiceInterface  myService = (MyServiceInterface)
context.getService(serviceReference);
```

Der Methode getService wird die gewünschte ServiceReference übergeben. Als Rückgabewert erhält man das konkrete OSGi Service.

Wenn mehrere Versionen eines Bundles im Framework installiert wurden, die dasselbe Service bereitstellen, kann nur jenes Service abgefragt und verwendet werden, von dem auch die Packages importiert wurden.

Service verwenden

Zur Verwendung des Services, wird die OSGi Service Registry nicht mehr verwendet. Hier wird über das bereitgestellte Interface direkt auf das Service-Objekt zugegriffen.

Service freigeben

Wird von einem Client-Bundle ein Service nicht mehr benötigt, muss es wieder freigegeben werden. Das ist erforderlich, damit das Framework die Beziehungen zwischen

den Service Bereitstellern und den Service Konsumenten konsistent verwalten kann. Zur Freigabe eines Services muss innerhalb eines Bundles die Methode `unget()` aus dem BundleContext verwendet werden, wodurch die Service-Abhängigkeit der beteiligten Bundles aufgelöst wird.

Für detaillierte Beschreibungen siehe [11].

2.1.6 Dynamische Services

Eine herausragende Eigenschaft von OSGi ist die Fähigkeit, Services dynamisch registrieren bzw. deregistrieren zu können. Anders als beispielsweise bei Softwaresystemen, die JNDI verwenden – bei denen davon ausgegangen wird, dass die Datenquelle während der Laufzeit des Programms permanent verfügbar ist – ergeben sich durch die gewonnene Dynamik in OSGi neue Problemstellungen:

1. Was ist zu tun, wenn ein Service an der Service Registry abgefragt wird, das (noch) nicht registriert wurde?
2. Was ist zu tun, wenn ein Service bereits konsumiert wurde, dieses aber nicht mehr existiert?

Grundsätzlich muss in einer OSGi Umgebung aufgrund der gebotenen Dynamik immer damit gerechnet werden, dass ein Service nicht oder nicht mehr in der OSGi Service Registry zu finden ist.

Falls ein Service (noch) nicht an der Service Registry angemeldet wurde, liefert der Aufruf der Methode `getServiceReference()` den Rückgabewert `null`. Dieser Fall muss immer überprüft werden.

Falls man bereits eine `ServiceReference` zugeteilt bekommen hat, das Service aber nicht mehr verfügbar ist, erhält man beim Aufruf von `getService()` den Rückgabewert `null`. In diesem Fall könnte erneut eine ServiceReference von der Service Registry angefordert werden, denn möglicherweise steht dort wieder ein neues, adäquates Service bereit.

Als Reaktion auf ein nicht oder nicht mehr vorhandenes Service gibt es grundsätzlich drei Möglichkeiten:

1. Die Quittierung mit einer Exception oder null retournieren.
2. Eine gewisse Zeit abwarten und danach erneut das Service anfordern.
3. Der Client startet seine Arbeit erst dann, wenn das Service vorhanden ist.

Der erste Punkt ist einfach zu implementieren, daher wird darauf nicht weiter eingegangen. Für den zweiten Punkt benötigt der Client eine Implementation des `ServiceTracker`'s. Für den dritten ist zusätzlich eine Implementation eines `ServiceTrackerCustomizer`'s erforderlich. Diese werden im Folgenden vorgestellt.

Service Tracker

Eine Methode um auf die Änderungen von Services in der OSGi Service Registry reagieren zu können, ist durch die Verwendung eines Service Tracker's möglich. Der Konsument eines Services benutzt bzw. erweitert dazu die Klasse `ServiceTracker` aus dem Package `org.osgi.util.tracker` und instantiiert die Klasse mit den gewünschten Suchkriterien für ein Service. Im folgenden Beispiel sollen Services, die das Interface `myServiceInterface` implementieren beobachtet werden:

```
myServiceTracker = new MyServiceTracker(context,
                        myServiceInterface.class.getName(), null);
```

Mit dem Aufruf von `myServiceTracker.open()` wird der ServiceTracker aktiviert. Analog dazu wird zur Deaktivierung des ServiceTrackers die Methode myServiceTracker.close() aufgerufen.

Der ServiceTracker übernimmt die Beobachtung von Zustandsänderungen in der OSGi Service Registry und kümmert sich um das Abfragen und Freigeben von der interessierenden Services.

Beispiel für den Konsum eines Services über die Verwendung eines ServiceTrackers:

```
MyServiceInterface myServiceInterface = (MyServiceInterface)
myServiceTracker.getService();
```

Der Client registriert eine Veränderung des beobachteten Services, sobald dieser Code aufgerufen wird.

Genauere Angaben zur Verwendung des Service Trackers sind aus [9] und [11] zu entnehmen.

ServiceTrackerCustomizer

Um über die Änderung eines OSGi Service-Zustandes sofort informiert zu werden, muss das Interface `ServiceTrackerCustomizer`, über den ServiceTracker, implementiert werden. Dieses Interface enthält Callback-Methoden (`addingServie()`, `modifiedService()` und `removedService()`), die nach einem Zustandswechsel eines Services vom Framework aufgerufen werden. Dabei ist darauf zu achten, dass der ServiceTracker Thread-Safe ist, das bedeutet, dass auch die Callback-Methoden Thread-Safe implementiert werden müssen.

Whiteboard-Pattern

Das Whiteboard-Pattern kann im OSGi Framework als Ersatz für das klassische Observer-Pattern [16] verwendet werden. Der wesentliche Unterschied bestehen darin, dass sich beim Whiteboard-Pattern der Event-Beobachter nicht selbst bei der Event-Quelle über `addEventListener()` bzw. `removeEventListener()` an- bzw. abmelden muss, sondern dass dafür die OSGi Service Registry genutzt wird. Dadurch wird Codierarbeit reduziert [9].

2.1.7 Fragment Bundles

Mit Hilfe von Fragment Bundles, die auch als Fragments bezeichnet werden, können die in Abschnitt 2.1.2 vorgestellten OSGi Bundles durch Klassen und Ressourcen erweitert werden. Fragments bieten eine Möglichkeit, dem Prinzip „Offen für Erweiterungen, geschlossen für Änderungen" (vgl. [15]) nachzukommen. Z.B. können über Extention-Points verschiedene Ausprägungen eines Produktes realisiert werden, wobei die speziellen Ausprägungen in Fragments implementiert werden und der generische Teil im OSGi Bundle.

Fragments haben in OSGi folgende Eigenschaften:

- Sie sind keine selbstständigen OSGi Bundles, d.h. sie haben keinen eigenen Lebenszyklus und keinen eigenen Activator.
- Das Host Bundle, welches durch ein Fragment erweitert wird, muss nichts über das Fragment wissen.

Um ein Fragment Bundle in OSGi zu realisieren muss es im MANIFEST.MF folgendermaßen spezifiziert werden:

```
Fragment-Host: A;bundle-version="3.0.0"
```

Mit diesem Manifest Header wird mitgeteilt, dass das Fragment das Bundle A in der Version „3.0.0" erweitert.

Zusätzlich kann ein Fragment noch weitere Manifest Header einbinden, wie `Import-Package`, `Export-Package` oder `Require-Bundle`. Das Host-Bundle nimmt dann alle vom Fragment benötigten oder bereitgestellten Ressourcen in die eigene Beschreibung auf.

Für nähere Informationen sei auf [9] und [11] verwiesen.

2.1.8 Security

Das auf dem JAVA-Security-Model [14] basierende Sicherheitskonzept beschreibt die Ausführungsberechtigung in der Modul-, Lifecycle- und Serviceschicht.

Da die Unternehmensstruktur und somit auch die Organisation von Applikationen sehr unterschiedlich sein kann, divergieren auch die Deployment-Varianten einer Anwendung. Im Folgenden werden mögliche Deployment-Varianten dargestellt, welche durch OSGi unterstützt werden:

- *Fall 1 – geschlossenes System*: Der Entwickler beschreibt die benötigten Rechte für ein Bundle in der Datei `OSGI-INF/permission.perm`. Der Administrator begutachtet das Bundle und installiert es, wenn es kein Sicherheitsrisiko birgt. Das OSGi Framework ist danach dafür verantwortlich, dass die Beschränkungen des Bundles nicht übertreten werden.
- *Fall 2 – Offener Deployment-Kanal*: Der Entwickler gibt wieder die Datei `OSGI-ING/permission.perm` an. Der Administrator begutachtet die geforderten Berechtigungen und signiert das Bundle. Danach stellt er das Bundle zum Gebrauch für den Enduser bereit, der seinerseits die Korrektheit der Signatur prüfen kann.
- *Fall 3 – Delegationsmodell*: Hier stellt der Administrator einem Anbieter-Unternehmen ein Zertifikat aus, mit dem der Anbieter nun sein Produkt (Bundle) signiert. Der Administrator kann nun auf Basis der Zertifikate Rechte vergeben [9].

Lokale Berechtigungen

Über die Vergabe von lokalen Berechtigungen durch den Entwickler wird das Deployment in geschlossenen Systemen realisiert. Der Administrator begutachtet die geforderten Rechte und installiert das Bundle, wenn alle Berechtigungsanforderungen erfüllt sind.

Bei der Entwicklung des Bundles muss in der Datei `OSGI-INF/permission.perm` festgelegt werden, welche Klassen, Ressourcen oder Services zur Ausführung benötigt werden. Wenn die Datei `OSGI-INF/permission.perm` nicht existiert, werden dem Bundle implizit alle Berechtigungen zugeteilt. Die Beschreibung der Berechtigungen in `OSGI-INF/permission.perm` hat in einer speziellen Syntax zu erfolgen. Siehe Kapitel 2 in [11].

Beispiel einiger Berechtigungen:

```
(org.osgi.framework.PackagePermission „org.foo.interface“
                    „IMPORT“)

(org.osgi.framework.ServicePermission
                    „org.foo.interface.MyService“ „get“)

(org.foo.internal.MyPermission „service“)
```

Diese drei Zeilen bedeuten Folgendes:

1. Zeile: Das Bundle erklärt, dass es das Package org.foo.interface importiert.

2. Zeile: Das Bundle benötigt die Berechtigung, „`MyService`“ aus der OSGi Service Registry anzufordern.

3. Zeile: Das Bundle benötigt die Berechtigung, das OSGi Service „`service`“ zu konsumieren.

Globale Berechtigungen

Bei dieser Variante (Delegationsmodell) vergibt der Administrator pauschal globale Berechtigungen an eine Gruppe von Bundles, die bestimmte Voraussetzungen erfüllen. Dabei wird ein Tupel, das sich aus einer Bedingung und der dazugehörigen Berechtigung zusammensetzt, definiert.

Für die Verwaltung von Berechtigungen verwendet der Administrator den Conditional Permission Admin Service, der Teil der OSGi Core Spezifikation ist. Darin sind zwei Standardbedingungen festgelegt:

- `org.osgi.service.condpermadmin.BundleSignerCondition`: Damit werden Berechtigungen aufgrund der Signatur (Zertifikat) vergeben.
- `org.osgi.service.condpermadmin.BundleLocationCondition`: Damit werden Berechtigungen aufgrund des Installationsortes vergeben.

Nachfolgend wird eine Möglichkeit dargestellt, wie globale Berechtigungen in OSGi implementiert werden können:

```
ConditionInfo conditionInfo=
  new ConditionInfo(„[org.osgi.service.condpermadmin.“ +
  „BundleLocationCondition \“*/A*\“]“);


PermissionInfo packagePermissionInfo=
  new PermissionInfo(„(org.osgi.framework.PackagePermission“ +
  „\“org.foo.interface\“ \“IMPORT\“)“);
PermissionInfo servicePermissionInfo=
  new PermissionInfo(„(org.osgi.framework.PackagePermission“ +
  „\“org.foo.interfaceMyService\“ \“get\“)“);


PermissionInfo translationPermissionInfo=
  new PermissionInfo(„(org.foo.internal.MyPermission“ +
  „\“service\“)“);


ConditionalPermissionAdmin admin = MyLocalCondPerAdmin();


admin.addConditionalPermissionInfo(
  new ConditionInfo[]{conditionInfo},
  new PermissionInfo[]{packagePermissionInfo,
        servicePermissionInfo, translationPermissionInfo}
);
```

In diesem Beispiel werden dieselben Berechtigungen verteilt, wie sie im Abschnitt Lokale Berechtigungen definiert wurden. Über die Objekte des Typs `ConditionInfo` werden Bedingungen gesetzt, während über `PermissionInfo` die Berechtigungen festgelegt werden. `ConditionalPermissionAdmin` ist ein `interface`, über das die Bedingungen mit den Berechtigungen gesetzt, abgefragt oder wieder aufgelöst werden können. Im obigen Beispiel werden über die Methode `addConditionalPermissionInfo()` die Bedingungs-Berechtigungs-Tupels gebildet.

Das exemplarische Codefragment muss in ein Bundle implementiert werden, das mit allen Berechtigungen ausgestattet ist, da dieses Bundle die Berechtigungen aller anderen Bundles verwaltet.

Für weitere Informationen siehe [9], [11] und [14].

2.1.9 OSGi Standard Services

Auf die Core Specification aufgesetzt, wurden von OSGi standardisierte Services definiert. Der Entwickler hat die Möglichkeit, diese (u.a. Logging Service, Services zur administrativen Unterstützung oder Deklarative Services) für seine Zwecke zu nutzen.

Im Folgenden sollen exemplarisch *Deklarative Services* herausgegriffen werden, weil diese zum Verständnis von Spring-DM bzw. SpringSource dm Application Server, die in Abschnitt 2.4 untersucht werden, beitragen.

Für eine ausführliche Beschreibung aller Services sei auf [12] verwiesen.

2.1.10 Declarative Services

Declarative Services sind ein Ersatz für die in Abschnitt 2.1.5 beschriebenen OSGi Services. Declarative Services werden als Komponenten implementiert. Eine Komponente besteht aus einer Komponentenklasse und einer Komponentenbeschreibung. Die OSGi Service Component Runtime regelt je nach Bedarf den Zustand der Komponenten. Service Komponenten haben im Gegensatz zu OSGi Services den Vorteil, dass sie unter anderem die Komplexität der dynamischen Softwaremodelle verringern, weil Service Components nur dann instantiiert werden, wenn alle benötigten Ressourcen vorhanden sind. Außerdem werden Systemressourcen gespart, wenn die Komponenten erst bei Bedarf geladen werden.

Komponentenklasse

Eine Komponentenklasse wird als POJO implementiert. Die Komponentenklasse muss einen Default-Konstruktor beinhalten. Ansonsten können noch optional Callback()-Methoden implementiert werden. Diese müssen allerdings eine definierte Signatur aufweisen (siehe [12]).

Komponentenbeschreibung

Die Beschreibung der Komponente erfolgt über eine XML-Datei, auf welche mit dem Manifest Header Service-Component verwiesen werden muss. Im Folgenden werden die wichtigsten Elemente zur Beschreibung einer Komponente anhand zweier Beispiele angeführt:

Mit der ersten Beschreibung wird von einer „someService"-Komponente ein Service mit beliebigen Eigenschaften (properties) publiziert. Das Service ist dabei ein Objekt der Klasse „ConcreteSomeService", welches über eine Schnittstelle nach außen, also über die OSGi Service Registry, bereitgestellt wird (provide interface):

```
<?xml version="1.0"?>
<component name="someService">
      <implementation class=
                        "org.bar.internal.ConcreteSomeService"/>
      <property name="translation.language" value="de"/>
      <property name="translation.country" value="DE"/>
      <service>
            <provide interface="org.bar.api.SomeService"/>
      </service>
</component>
```

Die zweite Beschreibung ist ein Beispiel für eine Komponente, welche „someService" konsumiert. In diesem Fall heißt die Komponente „clientComponent". Der Client-Komponente muss dazu die Schnittstelle des publizierenden Services bekannt sein, was durch die Angabe des `interface`-Elements erreicht wird. Über die Elemente `bind` und

unbind können optional Callback-Methoden angegeben werden, welche beim Verknüpfen bzw. Entkoppeln der Komponenten verwendet werden.

```
<?xml version="1.0"?>
<component name="clientComponent">
      <implementation class="org.foo.ClientComponent"/>
    <reference name="someService"
            interface="org.bar.api.SomeService"
            bind="setSomeService"
            unbind="unsetSomeService"
            />
</component>
```

Für detailliertere Ausführungen zu diesem Thema siehe [9] und [12].

2.2 Class Loading Mechanismen

Um besser zu verstehen warum OSGi als Basis für modulare und dynamische Softwaresysteme geeignet ist, werden in den nächsten Abschnitten unterschiedliche Class Loading Konzepte gegenübergestellt werden.

2.2.1 JAVA Standard Edition

Wie Abbildung 3 verdeutlicht, ist das Class Loading System in JAVA SE hierarchisch aufgebaut:

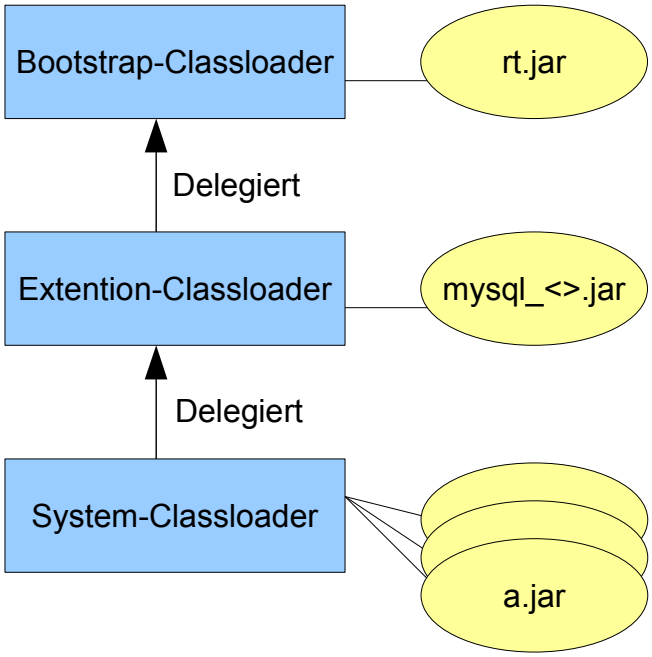

Abbildung 3: Class Loading System JAVA Standard Edition

Dabei sind drei Ebenen des Class Loader-Systems zu unterscheiden, die als Bootstrap-, Extention- und System Class Loader bezeichnet werden.

Der Bootstrap-Class Loader lädt Klassen, die bereits beim Hochfahren der JAVA Virtual Machine benötigt werden. Der Extention-Class Loader lädt Klassen, z.B. von Drittanbietern, die im Verzeichnis lib/ext abgelegt sind. Für alle anwendungsspezifischen Klassen ist der System-Class Loader verantwortlich. Alle für die Ausführung der Anwendung erforderlichen Klassen werden beim Starten der JVM durch das Argument -classpath in das Programm eingegliedert.

Das Suchen von Klassen im JAVA Class Loading System erfolgt durch das Delegieren von einer Ebene zur nächsthöheren. Das bedeutet, wenn eine Klasse benötigt wird, wird zuerst der System-Class Loader konsultiert. Kann dieser die Klasse im -classpath nicht finden, delegiert er die Anfrage für diese Klasse weiter in die Ebene des Extention-Class Loaders. Dieser durchsucht dann lib/ext. Wird die Klasse dann immer noch nicht gefunden, wird die Anfrage wieder eine Ebene nach oben weitergereicht. Wenn nun auch der Bootstrap-Class Loader die Klasse nicht finden kann, wird eine ClassNotFound-Exception geworfen.

In dem Falle, dass die Klasse gefunden wurde, wird sie geladen und die Anfrage ist damit beendet.

Vergleicht man nun das Potenzial der Versionierung von OSGi mit gewöhnlichen JAVA SE Anwendungen, werden die Schwächen des zweiteren Systems deutlich: Z.B. ist es dort nicht möglich, unterschiedliche Versionen einer Library zu verwenden. Problematisch ist auch, wenn mehrere Versionen derselben Library in verschiedenen Ebenen des Class Loading Systems vorkommen. Dann wird in einer Anwendung immer diejenige Klasse verwendet, welche in der Hierarchie als nächstes zu finden ist. Es kann sein, dass die erste gefundene Klasse von einer veralteten Version stammt.

Weitere Deteils siehe [14].

2.2.2 JAVA Enterprise Edition

Das Class Loading System der JAVA Enterprise Edition setzt auf das System von JAVA SE auf. Das Class Loading Konzept in JEE ist zwar ebenfalls hierarchisch aufgebaut, allerdings durch die JEE Spezifikation nicht so starr vorgegeben wie im Vergleich dazu JSE. Dies ist in Abbildung 4 skizziert. Der Grund dafür ist, dass serverseitige Anwendungen in ihren Ausprägungen stark divergieren können. Einige bestehen beispielsweise nur aus einfachen Servlets, andere wiederum verwenden EJB's usw. Alle

Komponenten können auf beliebige Art und Weise zu einer Komposition zusammengesetzt werden.

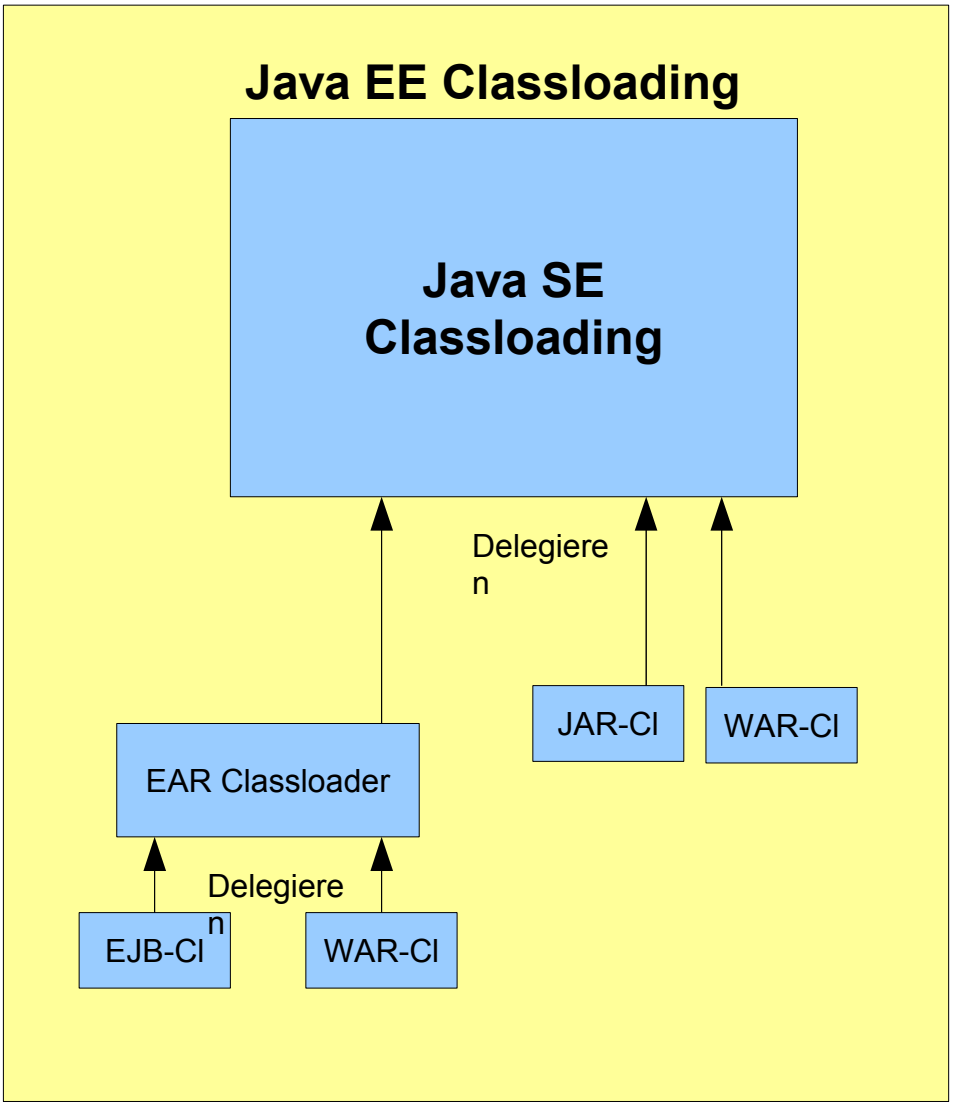

Abbildung 4: Class Loading System JAVA Enterprise Edition

Die Art, die JEE Anwendungen für die Verpackung von Komponenten bereitstellt, führt zu Problemen, wie sie im nachfolgenden Beispiel dargestellt werden (siehe [17]):

app2.ear:

META-INF/application.xml

ejb1.jar Class-Path: **ejb1_client.jar**

deployment descriptor contains:

<ejb-client-jar>ejb1_client.jar</ejb-client-jar>

ejb1_client.jar

ejb2.jar Class-Path: **ejb1_client.jar**

app3.ear:

META-INF/application.xml

```
ejb1_client.jar
ejb3.jar Class-Path: ejb1_client.jar
webapp.war Class-Path: ejb1_client.jar
WEB-INF/web.xml
WEB-INF/lib/servlet1.jar
```

Wie in diesem Beispiel ersichtlich, müssen alle benötigten Libraries für die fehlerfreie Ausführung der Anwendung im selben EAR verpackt werden. Hier wird deutlich, dass die Library `ejb1_client.jar` in beiden Anwendungen benötigt wird.

Die übliche Vorgehensweise ist, diese Library jeweils in das EAR zu verpacken. Dadurch entsteht häufig das Problem, dass die Anwendungen durch viele solcher Libraries aufgebläht werden und die Handhabung der Libraries unübersichtlich werden kann.

Eine weitere Methode wäre, Libraries, die von vielen Anwendungen benötigt werden (denkbar wären Logging- oder Security-Libraries), im Class Loading System einfach eine Ebene nach oben zu verschieben. Der Nachteil dieser Methode ist allerdings, dass nun die Deployment-Vorzüge des Application Server nicht mehr in Anspruch genommen werden können.

Für weiterführende Informationen zum Class Loading in JAVA EE siehe [17].

2.2.3 OSGi – Class Loading

Einen bedeutenden Unterschied zum herkömmlichen hierarchischen Class Loading-System in JAVA weist OSGi mit seinem graphenähnlichen Delegationsmodell für Class Loader auf (siehe Abbildung 6). Jedes Bundle hat dabei einen eigenen Class Loader. Da das Framework über die Informationen der Package-Abhängigkeiten verfügt, kann es gezielt die Class Loading-Delegation steuern.

In Abbildung 6 erkennt man beispielsweise, dass Bundle E eine Klasse aus Bundle A benötigt. Das wird dem Framework über den Import-Package Header im MANIFEST.MF mitgeteilt. Daraufhin wird veranlasst, dass die Class Loading Anfrage von Bundle E an den Class Loader von Bundle A weitergegeben wird. Dieser lädt nun die benötigte Klasse.

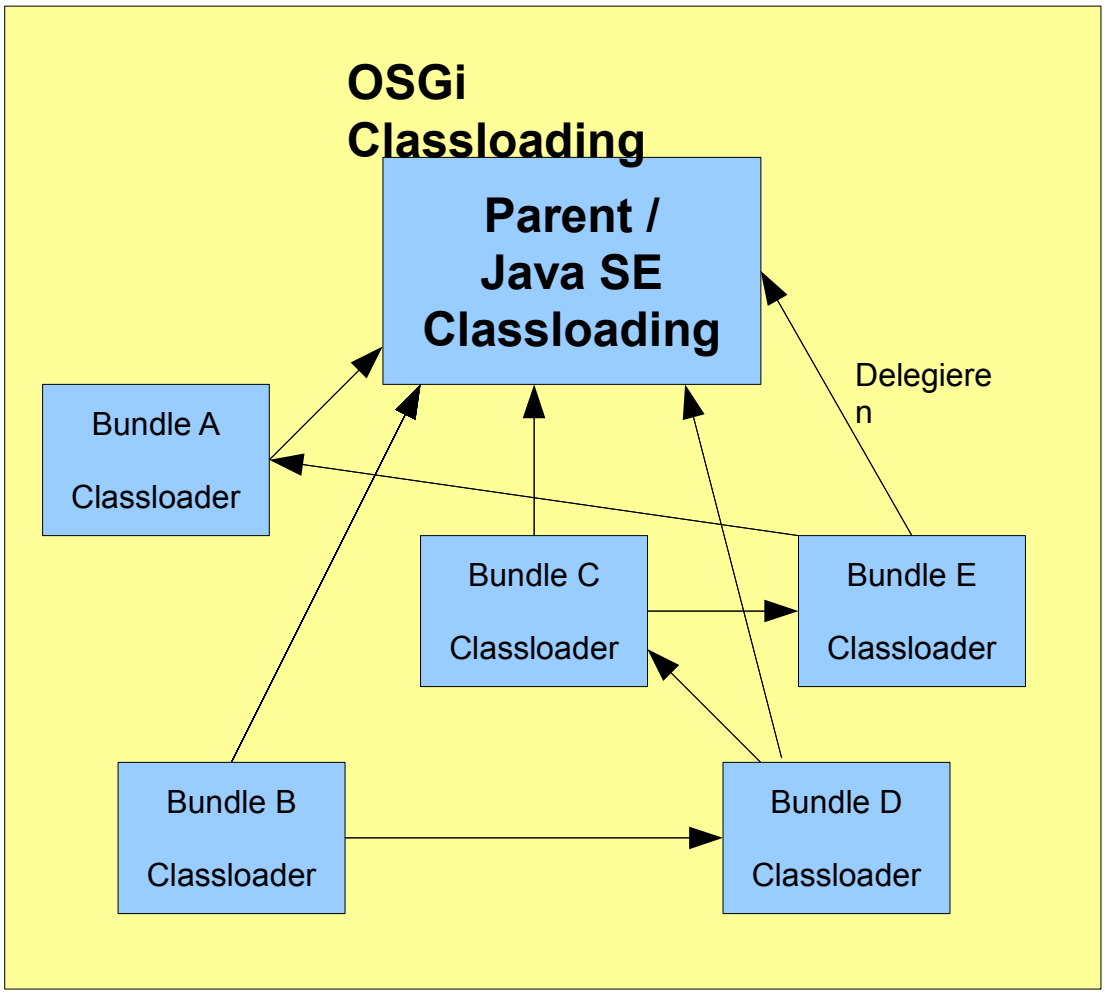

Abbildung 6: Class Loading Konzept - OSGi

Für Details zum Class Loading in OSGi siehe [11] und [9].

2.3 Das Potential von OSGi für JAVA Enterprise Edition

Wie in den vorherigen Abschnitten beschrieben, ist das OSGi Framework auf die Realisierung modular aufgebauter und dynamisch veränderbarer Anwendungen spezialisiert. Eine Vielzahl von Anwendungen aus unterschiedlichen Bereichen basiert bereits auf OSGi. OSGi unterstützt allerdings nicht die Anforderungen an verteilte Systeme, weil das Management von Klassen, Ressourcen und Objekten nur innerhalb einer JAVA Virtual Machine umgesetzt wurde.

In den folgenden Abschnitten wird das Potential von OSGi im Bereich der JAVA EE Anwendungen fokussiert. Exemplarisch wurden besonders populäre Applikationsserver, wie Sun GlassFish V3 prelude und JBoss 5.0, sowie der SpringSource dm Application Server für den Vergleich ausgewählt.

2.3.1 JBoss 5.0

JBoss ist einer der am weitesten verbreiteten Application Server, die frei erhältlich sind. JBoss 5.0 ist auf Basis von OSGi modular aufgebaut. Dadurch ist es besonders einfach, den Kern des Application Servers zu erweitern. Die Kernarchitektur von JBoss wird von der JBoss Community als JBoss Microcontainer bezeichnet. Ein wesentlicher Vorteil, den diese Server Architektur bietet, ist, dass nicht mehr zwingend alle Module von JBoss installiert werden müssen. Es können gezielt jene Server Module verwendet werden, welche für das Betreiben der jeweiligen Anwendung nötig sind. Für weitere Details zu JBoss Microcontainer und JBoss Application Server siehe [19].

Für den JEE Anwendungs-Entwickler sind aber durch die Erneuerungen der Application Server Architektur keine neuen Möglichkeiten hinzugekommen. JEE Anwendungen müssen gemäß der JEE Spezifikation erstellt werden. Die Fähigkeiten von OSGi bleiben für den JEE Entwickler verborgen.

2.3.2 GlassFish V3 Prelude

Ähnlich wie bei JBoss 5.0 verhielt sich die Entwicklung des Application Servers von Sun Microsystems. Die Architektur des Application Servers GlassFish V3 Prelude basiert auf dem OSGi Framework und kombiniert dieses mit HK2, einem eigens entwickelten Modulsystem. Damit besteht GlassFish aus Modulen, die es erlauben, den Application Server einfach über Add-on Komponenten erweitern zu können (siehe [20]).

Wie bei JBoss wird der JEE Entwickler auch durch GlassFish nicht durch OSGi Elemente unterstützt.

Für Informationen zu GlassFish siehe [21].

2.4 Spring

2.4.1 Spring Framework

Die Hauptmotivation für die Entwicklung des Spring Frameworks war, eine Alternative zur Entwicklung von J2EE Anwendungen zu schaffen. Insbesondere die Realisierung von EJB 2.x Komponenten war mit einem hohen zeitlichen Aufwand verbunden, weil man dazu gezwungen war, Schnittstellen zu implementieren, die man eventuell gar nicht benötigt. Hinzu kam die hohe Komplexität bei der Konfiguration der Komponenten, wodurch größere Anwendungen schnell unübersichtlich wurden.

Das Spring Framework sollte den Entwicklungsprozess von J2EE Anwendungen vereinfachen und somit beschleunigen. Einige der Kernprinzipien des Spring Frameworks werden im Folgenden angeführt, da sie in Weiterentwicklungen wie Spring-DM (siehe Abschnitt 2.4.2) oder SpringSource dm Application Server (siehe Abschnitt 2.4.3) immer noch von großer Bedeutung sind:

- Die Realisierung von Anwendungen auf Basis von bewährten objektorientierten Design- und Implementierungsmethoden, wie Vererbung und Polymorphie.
- Eine möglichst einfache Entwicklung des Middlelayers von JEE Anwendung soll durch das Framework unterstützt werden. Dies geschieht in Form von POJOs, welche in diesem Kontext als Spring Beans bezeichnet werden. Spring Beans kapseln nur den Code, der für die Realisierung der spezifischen Anforderungen einer Anwendung relevant ist. Es wird vermieden, dass der Entwickler Callback-Methoden bzw. Schnittstellen implementieren muss, welche für die Anwendung irrelevant sind.
- Inversion of Control (IoC) bzw. Dependency Injection: Dieses Konzept beschreibt die Fähigkeit des Containers, Abhängigkeiten (z.B. Datenquellen-Objekt) dem Client zu übergeben (injizieren), die sonst vom nutzenden Objekt selbst instantiiert werden müssten. Dabei wird der Kontrollfluss umgedreht.

- Aspektorientierte Programmierung: Ein Paradigma der objektorientierten Programmierung ist es, Anliegen zu trennen. Es gibt allerdings Fälle, in denen dieser Grundsatz durch die alleinige Anwendung der Objektorientierung nicht umgesetzt werden kann. Beispiele dafür sind Logging oder Security. Alle Klassen brauchen Methoden für Logging und Securitychecks, deshalb kann dieses Anliegen nicht getrennt werden. Solche Anliegen sind als Crosscutting Concerns bekannt. Die Aspektorientierung bietet für solche Anliegen einen Lösungsansatz, welcher durch Spring unterstützt wird.
- Das Testen der J2EE Anwendungen wurde durch das Spring Framework erheblich vereinfacht.

Für weitere Informationen siehe [22-26].

2.4.2 Spring Dynamic Modules

Spring Dynamic Modules (Spring-DM, früher Spring-OSGi) integriert das Spring Framework in eine OSGi Plattform. Das Ergebnis dieses Produkts ist eine Kombination von Spring Entwicklungs-Elementen, wie sie in Abschnitt 2.4.1 eingeführt wurden, mit den Vorteilen von OSGi (modularer Softwarearchitektur, Dynamik sowie Versionierung).

Im folgenden Abschnitt werden die wichtigsten Eigenschaften von Spring-DM dargestellt.

Bundles und ApplicationContext

OSGi Anwendungen basieren auf der Entwicklung von Bundles und OSGi Service (siehe Abschnitt 2.1). In Spring ist der Application Context die zentrale Stelle, um Spring Beans (Objekte) aus ihrer Beschreibung (meist einer XML-Datei) aufzunehmen und zu verwalten.

In Abbildung 7 ist dargestellt, wie OSGi und das Spring Framework in Spring-DM miteinander kombiniert werden.

In Spring-DM werden Spring-DM-Bundles entwickelt. Ein von Spring-DM zur Verfügung gestelltes Extender Bundle hat die Aufgabe, für jedes Spring-DM-Bundle einen eigenen Application Context zu erzeugen. Die Zuteilung des Application Contexts zum Spring-DM-Bundle erfolgt asynchron. Wenn ein Spring-DM-Bundle deinstalliert wird, wird auch der Application Context wieder zerstört. Wird ein neues Spring-DM-Bundle installiert, erzeugt das Extender Bundle dafür einen neuen Application Context.

Spring Beans werden über die Bundle-Grenze mittels der OSGi Service Registry als OSGi Services publiziert bzw. konsumiert (vgl. Abbildung 7).

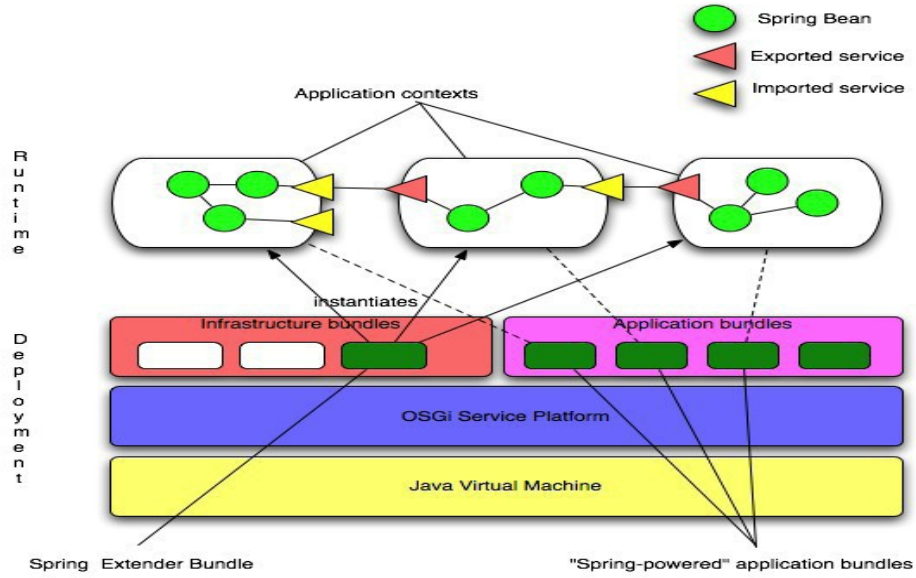

Abbildung 7: Zusammenspiel von Spring und OSGi durch Spring-DM (Quelle: [27])

Packaging

Das Extender-Bundle erkennt ein Bundle innerhalb des Frameworks als Spring-DM-Bundle entweder durch den zusätzlichen Manifest Header `Spring-Context` oder das Existieren des Verzeichnisses `META-INF/spring`, das die XML-Beschreibungen für Spring-DM beinhaltet.

Üblicherweise wird der Application Context mit Hilfe der Konfigurationen `<MODUL_NAME>-context.xml` und `<MODULE_NAME>-osgi-context.xml` erzeugt.

`<MODULE_NAME>-context.xml` beinhaltet Spring Bean Definitionen, wie sie im Spring Framework üblich sind. Diese Datei enthält keine OSGi-Spezifika.

`<MODULE_NAME>-osgi-context.xml` beinhaltet Spring Beans, die als OSGi Services in die Service Registry importiert bzw. exportiert werden sollen.

Runtime

Das folgende Beispiel zeigt, wie eine Spring Bean im *-context.xml beschrieben wird:

```
<bean name="mySimpleBean"
class="org.foo.internal.MyConcreteService" />
```

Die Klasse MyConcreteService wird zur Laufzeit vom ApplicationContext instantiiert und verwaltet. Um diese Spring Bean („mySimpleBean“) als OSGi Service bereitzustellen, ist zumindest die Angabe der Bean und der bereitgestellten Schnittstelle, im *-osgi-context.xml notwendig:

```
<service ref=“mySimpleBean“
interface="org.foo.interface.MyService" />
```

Die Spring Bean wird in diesem Beispiel unter der Schnittstelle „org.foo.interface. MyService“ an der OSGi Service Registry bereitgestellt und kann nun über die Angabe der Schnittstelle aus der OSGi Service Registry angefordert und verwendet werden:

```
<reference id=“myServiceOSGi“
interface=“org.foo.interface.MyService“/>
```

Das Element <reference> erzeugt, dabei wiederum eine Spring Bean, welches ein passendes Service von der OSGi Service Registry injiziert bekommt.

Für weitere Informationen zu Spring Dynamic Modules sei auf [27-28] verwiesen.

2.4.3 SpringSource dm Application Server

Im Gegensatz zu Spring Framework bzw. Spring-DM ist SpringSource dm Application Server ein vollwertiger Applikationsserver. Ein wichtiges Merkmal des dm Servers, wie er auch bezeichnet wird, ist, dass er nicht nur wie die Serverarchitekturen von JBoss und GlassFish auf OSGi aufsetzt, sondern dem JEE Entwickler die Möglichkeit bietet, eine OSGi-basierte Anwendung zu realisieren (siehe [29]).

Basistechnologien des dm Servers

Der Kern des dm Servers baut auf folgenden Technologien auf:

1. Spring Framework
2. OSGi
3. Spring-DM, als Brücke zwischen Spring Framework und OSGi

Spring-DM bildet das Herzstück des dm Server Kerns. Es baut auf die Equinox Platform auf und erweitert diese über Extender Bundles (siehe Abbildung 7). Dabei werden Vorteile von OSGi, wie Versionierung und Modularisierung, genutzt. Basierend auf Spring-DM werden Subsysteme installiert, welche Funktionalitäten für den dm Server beinhalten.

Anwendungen, die vom dm Server unterstützt werden, sind modular aufgebaut und werden daher als OSGi Bundles implementiert. Jedes Bundle hat dabei einen bestimmten

Zweck zu erfüllen (z.B. Web, batch oder web service). Jedes Bundle kann während der Laufzeit des dm Servers installiert, geändert oder deinstalliert werden.

Um den Anforderungen von serverseitigen Anwendungen gerecht zu werden, wurde die OSGi Umgebung durch den dm Server erweitert und ergänzt:

- In großen Anwendungen kann das explizite Importieren von JAVA Packages sehr komplex werden. Daher können Libraries erstellt werden, die Packages bündeln und anstelle einer Reihe von einzelnen Packages importiert werden können.
- In OSGi gibt es nur die Sichtweise auf Module als Bestandteile der Anwendung und keine Sicht auf die Gesamtanwendung.

Folgendes Beispiel soll eine Konfliktsituation für den zweiten Punkt illustrieren:

Ein Application Server beherbergt mehrere serverseitige Anwendungen. Jede dieser Anwendungen beinhaltet alle für sie benötigten Libraries und Bundles. Wenn nun zwei Anwendungen dasselbe Bundle verwenden (denselben `Bundle-SymbolicName` und dieselbe `Bundle-Version`), meldet OSGi beim Deployment der zweiten Applikation einem Fehler. Für die Lösung dieses Problems wurde das Deployment-Format PAR entwickelt. Mit diesem ist es möglich, die gesamte Anwendung zu verpacken und zu installieren.

Library Konzept

Um sehr lange Listen von Packages im Manifest Header `Import-Package` zu vermeiden, wird ein Library Konzept für OSGi eingeführt. Diese Libraries sind normale Text-Dateien, welche typischerweise die Dateiendung .libd haben.

Beispiel einer Library (aus [30]):

```
Library-SymbolicName: org.springframework.spring
Library-Version: 2.5.4
Library-Name: Spring Framework
Import-Bundle: org.springframework.core;version="[2.5.4,2.5.5)",
org.springframework.beans;version="[2.5.4,2.5.5)",
org.springframework.context;version="[2.5.4,2.5.5)",
org.springframework.aop;version="[2.5.4,2.5.5)",
org.springframework.web;version="[2.5.4,2.5.5)",
org.springframework.web.servlet;version="[2.5.4,2.5.5)",
org.springframework.jdbc;version="[2.5.4,2.5.5)"
```

Für die Verwendung einer solchen Library wird der proprietäre Manifest Header wie folgt eingebunden:

```
Import-Library: org.springframework.spring;version="[2.5.4, 3.0)"
```

Analog dazu ist die Verwendung von `Import-Bundle` möglich.

Provisioning Repository

Einen wesentlichen Unterschied zwischen den gängigen Application Servern und dem dm Server stellt das Provisioning Repository dar. Bei Standard JEE Anwendungen ist es üblich, alle Libraries der Anwendung bzw. alle für die Anwendung benötigten externen Libraries in die Anwendung mit zu verpacken. Beispielsweise ist das Verzeichnis `WEB-INF\lib` bei Servlets für die Aufnahme von exteren Libraries vorgesehen.

Der dm Server stellt ein Repository für externe Libraries zur Verfügung, um das Aufblähen der Libraries zu vermeiden bzw. einen zentralen Ablageort für solche Klassen und Ressourcen zur Verfügung zu stellen.

Das Repository ist per default in `%SERVER_HOME%\repository` lokalisiert und enthält drei Subverzeichnisse: `bundles`, `libraries` und `installed`. `Bundles` enthält die am Server

installierten Bundles, `libraries` enthält die am Server installierten Libraries, die wieder auf die Bundles verweisen und `installed` enthält die vom Server zur Laufzeit benötigten Bundles und Libraries. Das Verzeichnis `bundles` ist wiederum in `ext`, `subsystems` und `usr` aufgeteilt. `subsystems` ist dabei nur für die serverinterne Nutzung vorgesehen. Die anderen beiden stehen dem Anwender für die Installation von Drittanbieter-Libraries (`ext`) bzw. für die Installation von Libraries des Endanwenders zur Verfügung (`usr`). Ganz ähnlich ist auch das `libraries`-Verzeichnis strukturiert [31].

Deployment-Pakete

SpringSource dm unterstützt mehrere Deployment-Paketformen:

- Normale OSGi Bundles
- Standard WAR
- WAR Shared Library
- WAR Shared Services
- Web Modules
- PAR

Nachfolgend werden die verschiedenen Deployment-Varianten genauer erläutert:

Normale OSGi Bundles: Da der dm Server auf Equinox aufsetzt, können gewöhnliche OSGi Bundles installiert werden. Solche Bundles können Funktionalitäten beherbergen, die allen auf dem Application Server installierten Anwendungen zur Verfügung stehen.

Standard WAR: Das Web Application Archive (WAR) besitzt eine spezielle, standardisierte Dateistruktur, die in einer JAR-Datei komprimiert ist. Zur Beschreibung der Applikation dient ein Deployment Deskriptor (web.xml). Libraries von Drittanbietern werden im Verzeichnis `WEB-INF\lib` abgelegt (siehe Kapitel SRV.9.5 [32]). Standard WAR's werden vom dm Server zur Laufzeit in ein OSGi Bundle umgewandelt und auf Tomcat installiert.

WAR shared library: Ausgehend von Standard WAR wird hier die Fähigkeit von OSGi genutzt, Packages exportieren und importieren zu können. Dieser Mechanismus wird beispielsweise dazu verwendet um externe Libraries aus einem Repository importieren zu können. Das Verzeichnis `WEB-INF\lib` fällt bei dieser Variante weg, wodurch ein Aufblähen des Deployment-Artefakts vermieden wird.

WAR shared services: Diese Variante baut auf dem Konzept der WAR shared libraries auf und nützt zusätzlich OSGi Services. Ähnlich wie bei Spring-DM, werden OSGi Services deklarativ in die OSGi Service Registry publiziert (`<osgi:service [...]/>`) und über den Application Context konsumiert (`<osgi:reference [...]/>`).

Web Module: Ausgehend von WAR shared services stellen Web Moduls noch zusätzliche Vereinfachungen für Spring MVC Entwickler zur Verfügung. Zur Konfiguration des `DispacherServlet`'s und `FilterMapping`'s werden neue Manifest Header eingeführt. Beispielsweise `Web-DispacherServletUrlPatterns` oder `Web-FilterMappings`. Auf die Konzepte von MVC und speziell Spring MVC wird hier nicht näher eingegangen. Hierfür sei auf [15] und [23] verwiesen.

Abbildung 8 skizziert, wie durch die Migration von Standard WAR zu Web Modulen sukzessive OSGi Features, für die Entwicklung von JEE-Anwendungen, gewonnen werden.

PAR: Für den dm Server gibt es ein neues Verpackungsformat, welches vergleichbar mit dem EAR aus dem JEE-Standard ist (siehe Kapitel EE 8.1.2 [17]). Die einzelnen Module, die eine Anwendung ergeben, werden im PAR zusammengefasst. Dadurch ist der Sichtbarkeitsbereich der Module auf die Anwendung begrenzt. Durch die Verpackung der Anwendung in ein PAR wird eine übergeordnete Sicht auf die Anwendung eingeführt. In technischer Hinsicht ist ein PAR eine JAR-Datei, die alle Module einer Anwendung bündelt. Mit einem PAR ist es möglich, die Gesamtapplikation zu installieren, aktualisieren bzw. deinstallieren.

PAR's enthalten spezielle Manifest Header, wie zum Beispiel `Application-SymbolicName` und `Application-Version`.

Für weitere Informationen siehe [31].

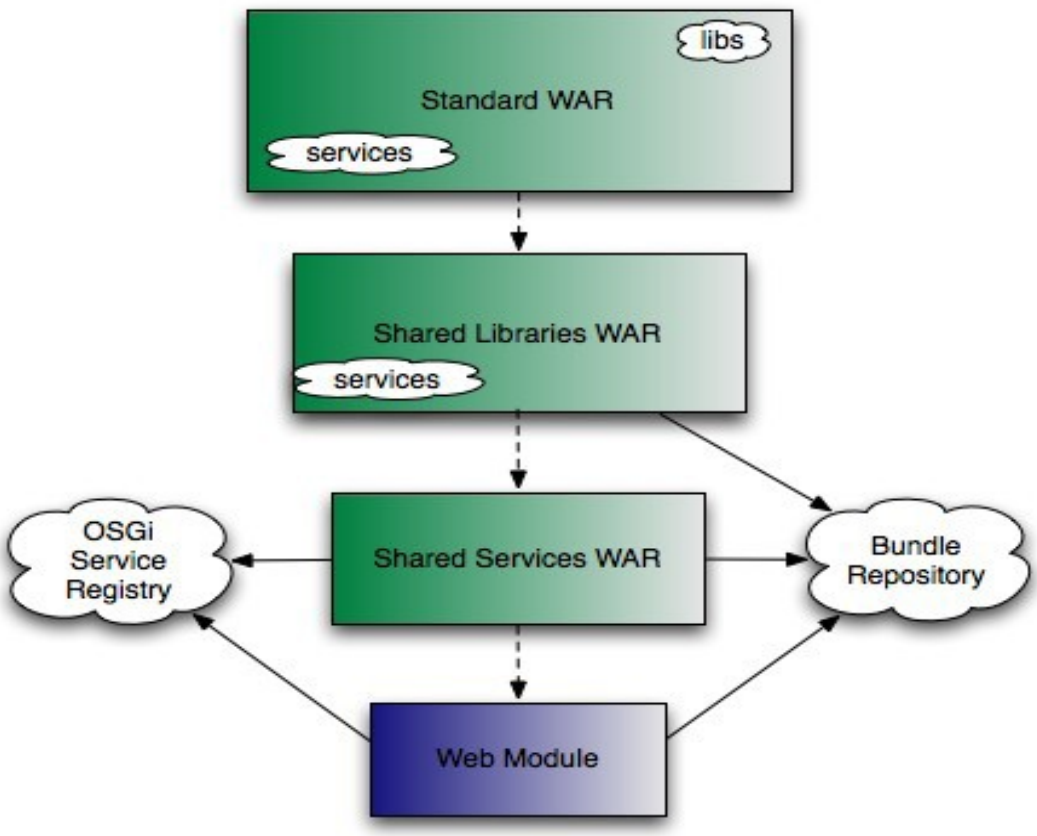

Abbildung 8: Migration einer Webanwendung auf proprietäre Deploymentformen des dm Server (Quelle: [30])

Deployment Varianten

Der dm Server bietet die Möglichkeit, Anwendungen während der Laufzeit installieren, ändern und deinstallieren zu können. Dazu werden zwei verschieden Deployment-Methoden angeboten.

Über die Admin Console können einerseits manuell PAR-, WAR- und JAR-Dateien auf dem Server installiert werden. In diesem Fall übernimmt der Server die Auflösung der Package-Abhängigkeiten in der installierten Anwendung.

Andererseits unterstützt der dm Server eine Hot-Deployment-Methode. Hier wird die zu installierende Anwendung einfach in das Verzeichnis `%SERVER_HOME%\pickup` kopiert. Der Server nimmt diese Pakete automatisch auf und kopiert sie in das `work`-Verzeichnis. Dabei ist es wichtig, die Abhängigkeiten der Pakete über die Deployment-Reihenfolge zu berücksichtigen, d.h. es sollten nicht alle Pakete mit einem Kopierbefehl in das Verzeichnis kopiert werden, wenn sie voneinander abhängen, da der Server die Abhängigkeiten sonst möglicherweise nicht auflöst.

Analog dazu existieren die beiden Methoden auch für das Undeployment.

2.5 Newton

2.5.1 Überblick

Newton ist ein Open Source Framework, das für verteilte, komponentenbasierte, service-ortientierten Anwendungsmodelle konzipiert wurde. Die Ziele von Newton sind:

- Unterstützung dynamischer, verteilter Systeme [6-7].
- Einfache Entwicklung von Komponenten in Form von POJOs.

2.5.2 Technologien / Spezifikationen

Als Kern für Newton werden folgende Technologien verwendet:

- OSGi
- JINI
- Service Component Architecture (SCA)

OSGi

Die wichtigsten Charakteristika von OSGi wurden bereits in Abschnitt 2.1 erwähnt. In Newton spielt OSGi eine zentrale Rolle. Unter anderem wird mit Hilfe der OSGi Service Registry die Verknüpfung von Services innerhalb einer JVM, d.h. einer Newton Instanz erledigt. Dieser Prozess wird in Newton als binding bezeichnet.

JINI

Jini definiert eine serviceorientierte Anwendungsarchitektur, welche sich als Ziel setzt, skalierbare, leicht veränderbare verteilte Systeme durch ihr Programmiermodell zu unterstützen.

In Jini wird eine Service Registry verwendet, über die sich im Netzwerk verteilte Services gegenseitig finden können. Newton verwendet diese Technologie, um verteilte Komponenten zu verknüpfen (siehe Binding) [33].

Service Component Architecture

Unter SCA werden unterschiedlichste Spezifikationen zur Standardisierung serviceorientierte Architekturen zusammengefasst.

Die SCA wird von einem Zusammenschluss aus Industriepartnern geleitet, welche unter dem Namen Open Service Oriented Architecture Collaboration bekannt sind.

SCA baut auf bewährten Ansätzen, wie z.B. Web Services oder RMI, auf und erweitert diese. SCA spezifiziert Service-Komponenten, die zu Service Kompositionen zusammengefasst sind [34-36]. Newton bietet eine dynamische Implementation der SCA Spezifikation.

2.5.3 Das Komponentenmodell in Newton

Ein SCA Composite ist das kleinste Deployment Artefakt in Newton. Kompositionen wiederum bestehen aus Komponenten. Diese Komponenten implementieren Services, welche die Komposition bereitstellt. Über Referenzen können Services anderer Kompositionen verwendet werden. In Abbildung 9 wird der Aufbau einer SCA Komposition illustriert.

Kompositionen setzen sich einerseits aus den Klassen und Ressourcen zusammen, andererseits beinhalten sie Meta-Informationen, die unter anderem beschreiben, wie die Services verknüpft werden.

Die Beschreibung einer Komposition setzt sich typischerweise aus einem Composite Template Descriptor und einem Composite Instance Descriptor zusammen.

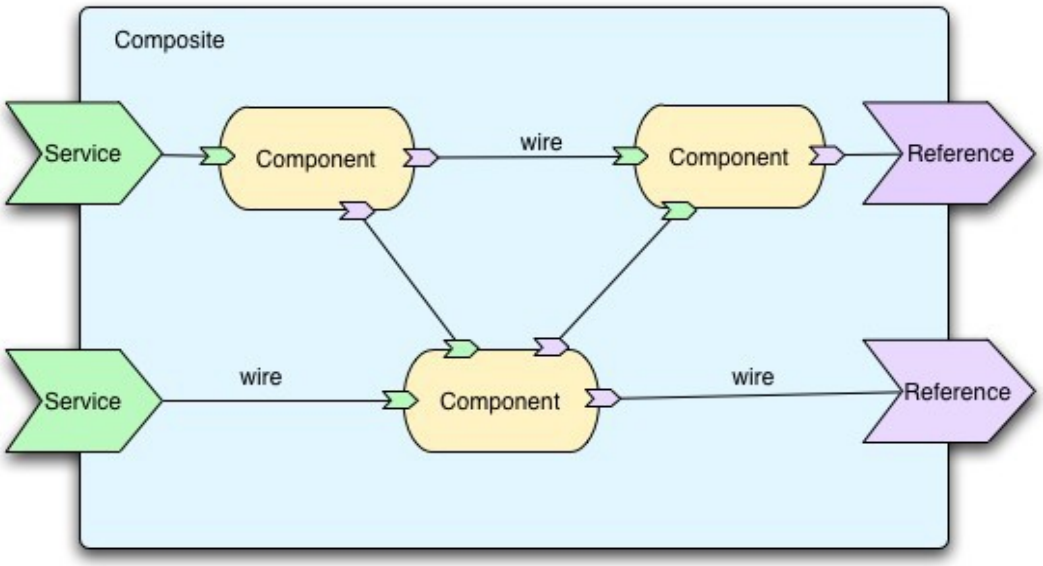

Abbildung 9: SCA Kompsition in Newton (Quelle: [37])

Der Composite Template Descriptor dient zur Beschreibung eines generischen Kompositionstyps. Das nachfolgende Beispiel illustriert einen Composite Template Descriptor:

```
<?xml version="1.0"?>
<composite name="myInfoComposite">
  <description>My expample Composite</description>

  <service name="give-info">
      <interface.JAVA interface= "org.foo.interface.InfoWriter"/>
      <binding.rmi/>
  </service>

  <reference name="get-info" multiplicity="1..1">
    <interface.JAVA interface= "org.foo.interface.InfoReader"/>
    <binding.osgi filter=""/>
  </reference>

  <component name="info">
    <reference name=“get-info“/>
    <implementation.JAVA.callback impl= "org.foo.impl.ConcreteInfo"/>
  </component>

  <wire>
    <source.uri>info</source.uri>
    <target.uri>give-info</target.uri>
  </wire>
```

```
    <wire>
      <source.uri>get-info</source.uri>
      <target.uri>info/get-info</target.uri>
    </wire>
  </composite>
```

In diesem Composite Template Descriptor wird eine Kompositionsvorlage namens myInfoComposite gebildet. Die Komposition stellt ein remote Service bereit (give-info) und konsumiert gleichzeitig ein lokales Service (get-info) einer anderen Komposition. Das Service und die Referenz werden jeweils über die wire-Elemente mit der Komponente (info) verknüpft.

Für die Beschreibung einer speziellen Ausprägung einer Komposition dient der Composite Instance Descriptor. Mit dieser Beschreibung werden die zuvor beschriebenen generischen Kompositionstypen je nach Bedarf überschrieben und erweitert. Beispiel für einen Composite Instance Descriptor:

```
<?xml version="1.0"?>
  <composite name="myInfoComposite">
    <bundle.root bundle="info-bundle" version="1.0"/>
    <include name="myInfoComposite"/>
</composite>
```

In diesem Beispiel wird der Kompositionstyp des myInfoComposite erweitert, was durch das include-Element geschieht. Bei der Installation einer SCA Komposition auf einer Newton Instanz wird ein OSGi Bundle erzeugt und verwaltet (in diesem Beispiel „info-bundle“ in der Version „1.0“).

2.5.4 Installation eine Komposition

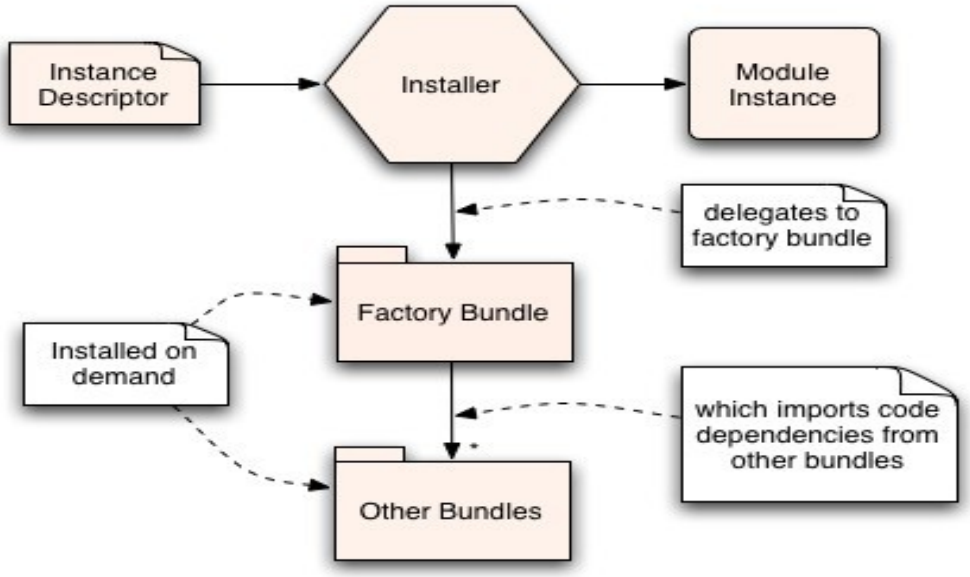

Abbildung 10: Installationsprozess von Kompositionen in Newton (Quelle: [37])

Wie in Abbildung 10 ersichtlich ist, wird zur Installation eines Deployment-Artefakts der Composite Instance Descriptor an den Newton Installer übergeben. Dieser delegiert den Installationsprozess weiter an ein Factory Bundle, welches alle benötigten Klassen und Ressourcen für eine Komposition lädt bzw. importiert und daraus ein OSGi Bundle erzeugt. Werden daraufhin weitere Instanzen der Komposition benötigt, werden die bestehenden Klassen und Ressourcen des vorhandenen OSGi Bundles verwendet.

Während der Installationsphase werden die benötigten Implementationen der Komponenten instantiiert und danach von Newton gemanagt.

Wird der Composite Instance Descriptor aus der Newton Instanz wieder gelöscht, wird das Factory Bundle veranlasst die Komposition zu entfernen. Das Factory Bundle löscht das erzeugte OSGi Bundle aber erst dann, wenn keine Instanzen dieses Kompositionstyps mehr installiert sind.

2.5.5 Lebenszyklus von Kompositionen

Eine Komposition durchläuft in Newton folgende Lebenszyklen:

Lebenszyklus-Zustände einer SCA Komposition
RESOLUTION
DOWNLOAD
VALIDATE
ACTIVATE
RUNNING
DEACTIVATE
GARBAGE COLLECT

Tabelle 2: Lebenszyklus Zustände einer SCA Komposition

Resolution: Newton versucht, die Ressourcen-Abhängigkeiten aufzulösen.

Download: Alle Bundles, die noch nicht auf der lokalen Newton Instanz vorhanden sind, werden vom CDS (siehe 2.5.7) heruntergeladen.

Validate: Hier werden sicherheitstechnische Überprüfungen durchgeführt. Ein Guard beobachtet anhand von extern vorgegebenen Bestimmungen das Verhalten einer Komposition und hält die Aktivierung zurück, wenn Sicherheitsregeln gebrochen wurden.

Activate: In diesem Zustand werden die benötigten Objekte instantiiert, Services publiziert und Referenzen mit den Services verknüpft.

Running: In diesem Zustand stehen die publizierten Services anderer Kompositionen zur Verfügung. Es werden periodische Sicherheitschecks durchgeführt, die zu einer Deaktivierung der Komposition führen, wenn die Sicherheitsvorschriften nicht mehr eingehalten werden.

Deactivate: Die Komposition wird mit sämtlichen Services und Referenzen während der Laufzeit von Newton entfernt. Allerdings sind noch alle für die Ausführung der Komposition benötigten OSGi Bundles in der Newton Instanz vorhanden. Erst wenn ein Factory Bundle keine Instanz mehr zu verwalten hat, wird dieser Zustand verlassen und die Bundles zum Löschen freigegeben.

Garbage Collect: Angelehnt an den Garbage Collector Mechanismus von JAVA, werden sich in diesem Zustand befindliche Bundles von der Newton Instanz entfernt, weil keine Kompositionen mehr darauf referenzieren.

2.5.6 Binding

Das Beispiel zum Composite Template Descriptor aus Abschnitt 2.5.3 zeigt, wie Services und Referenzen deklariert werden. Bei der Installation einer Komposition werden diese über einen Prozess, der als Binding bezeichnet wird, verknüpft.

Binding innerhalb einer Newton Instanz wird dabei über die OSGi Service Registry realisiert. Dazu muss das Service oder die Referenz mit dem Element <binding.osgi/> annotiert werden. Für das remote binding nützt Newton die Technologie von Jini. Services und Referenzen die nicht in der selben JVM vorliegen müssen mit <binding.rmi/> markiert werden.

Ähnlich wie bei OSGi Services werden auch bei Newton die Services über das Key-Value-Prinzip mit beliebigen Eigenschaften annotiert. Referenzen werden mit einem auf LDAP-basierenden Filterstring versehen, um eine differenziertere Selektion für die benötigten Services zu ermöglichen.

Über einen Callback Lebenszyklus werden die benötigten Referenzen mit den passenden, verfügbaren Services verknüpft. Dabei werden Callback-Methoden der Referenz-Komponenten vom Framework aufgerufen, welche eine spezielle Signatur aufweisen (add<requirement-name>). Dadurch werden die Services mit den Referenzen auf JAVA-Objekt-Ebene miteinander gekoppelt.

Analog dazu ruft das Framework bei der Entkopplung Callback-Methoden (remove<requirement-name>) auf.

Newton erlaubt dem Entwickler den Binding-Prozess nach seinen Vorstellungen zu erweitern. Dazu stehen die beiden Schnittstellen `ReferenceBindingFactory` bzw. `ServiceBindingFactory` zur Verfügung. Beide Factories erstellen `ServiceBinding` bzw. `ReferenceBinding` Objekte. Die Aufgabe eines ServiceBinding-Objekts ist es einerseits, Lebenszyklus-Änderungen des Frameworks zu beobachten, und andererseits eine externe Schnittstelle zu veröffentlichen.

Ein ReferenceBinding-Objekt beobachtet ebenfalls die Lebenszyklus-Änderungen des Newton Frameworks. Darüber hinaus stellt es einen Mechanismus bereit, um die Referenz über ein Proxy-Objekt mit dem benötigten Service zu verknüpfen.

2.5.7 Content Distribution Service (CDS)

Das CDS wird verwendet, um binäre Inhalte auf die einzelnen Newton Instanzen innerhalb einer Newton Fabric - der Zusammenschluss mehrerer Newton Instanzen wird als Newton Fabric bezeichnet - zu verteilen. Typischerweise handelt es sich bei diesen Inhalten um OSGi Bundles, die zentral im CDS abgelegt sind. Die Inhalte, die im CDS abgespeichert sind, werden unter speziellen Namen und Attributen abgelegt, z.B.:

```
org.foo.helloworld, version=1.0, provider=helloworld.org
```

Hierbei handelt es sich um das OSGi Bundle org.foo.helloworld in der Version 1.0, das durch helloworld.org in der CDS bereitgestellt wird.

Auf die Inhalte im CDS kann über eine URL (nur lesend) oder über die von Newton bereitgestellte JAVA Schnittstelle CDService (lesend und schreibend) zugegriffen werden.

2.5.8 Provisioning

Für die administrative Bereitstellung von Systemen, die auf Newton basieren, stehen der Newton System Manager und das Provisioning Service zur Verfügung.

Newton System Manager

Ein System Administrator erstellt bei dieser Variante einen System Descriptor, der vom Newton System Manager gelesen wird. In diesem Descriptor wird festgelegt, wie viele und welche Kompositionen wo in der Newton Fabric installiert werden. Der System Administrator kann einerseits ein statisches System definieren, d.h. Typen und Anzahl der Kompositionen werden fix vorgegeben. Andererseits kann über die Verwendung eines ReplicationHandler (siehe Replication Handler) die Anzahl der Kompositionen dynamisch je nach Bedarf variieren. Z.B. kann ein skalierbarer ReplicatorHandler verwendet werden, der automatisch pro Newton Instanz eine Komposition eines gewissen Typs installiert.

Ein vom Systemadministrator geänderter System Descriptor kann der Newton Instanz während der Laufzeit übergeben werden. Die Systemänderungen werden automatisch vom Framework abgeglichen.

Provisioning Service

Wie der Newton System Manager ist auch das Provisioning Service für die Verteilung und Installation von Kompositionen in einer Newton Fabric verantwortlich. Hier entfällt aber der manuelle Eingriff des Systemadministrators.

Wird eine zu installierende Komposition an das Provisioning Service übergeben, sendet dieses eine Hosting-Anfrage, welche die Definition der Komposition beinhaltet, an alle Newton Instanzen der Newton Fabric. Jede Newton Instanz eruiert daraufhin auf Basis eines Bewertungsschemas jene Kosten, die durch die Installation der Komposition entstehen und sendet die Angaben wieder zurück zum Provisioner.

Prinzipiell setzen sich die Kosten aus einer Gewichtung, die die Systemanforderungen der Komposition widerspiegelt, und einer Gewichtung, die die Systemkapazitäten beschreibt, zusammen. Eine Komposition wird mit sogenannte Contracts annotiert, in denen die Anforderungen für eine lauffähige Komposition festgelegt wird. In den sogenannten Features und Assessors wird dabei festgelegt, welche Leistungen das System bieten kann.

Der Provisioner installiert die Komposition auf jene Newton Instanz, welche die Hosting-Anfrage mit den geringsten Kosten quittiert.

Für die installierten Kompositionen kontrolliert das Provisioning Service periodisch die Kosten und gleicht das Softwaresystem bei Bedarf automatisch an geänderte Umgebungsbedingungen ab.

Replication Handler

Über die Schnittstellen ReplicationHandler und ReplicationHandlerFactory bietet das Newton Framework einen Erweiterungspunkt für die dynamische Skalierung von Kompositionen mit der Systemgröße. Der Entwickler muss über das klassische Abstract Factory-Design Pattern eine konkrete ReplicationHandlerFactory und den konkreten zu erstellenden ReplicationHandler implementieren. Die Implementation der ReplicationHandlerFactory wird über die OSGi Service Registry angemeldet und erzeugt ReplicationHandler-Objekte, basierend auf dem ReplicationDescriptor-Element im System Descriptor. Die Kompositionen werden durch die konkreten ReplicationHandler-Objekte erzeugt.

Für weiterführende Informationen siehe [16] und [37-38].

3 Resultate

3.1 OSGi

Vorteile

- Mit dem OSGi Framework ist es möglich, modular JAVA-Anwendungen zu entwickeln, was durch die Programmiersprache JAVA allein nicht unterstützt wird.
- Anwendungen auf Basis von OSGi unterstützen hochdynamische Anforderungen. Module können zur Laufzeit installiert, geändert oder deinstalliert werden. OSGi Services können dynamisch in der OSGi Service Registry bereitgestellt oder entfernt werden.
- OSGi unterstützt Softwareversionierung. In JAVA SE gibt es keine Versionierungsmechanismen.
- Durch die gewonnene Dynamik auf der Service-Ebene entstehen zwar neue Problemstellungen, die aber durch Hilfsmittel des Frameworks einfach gehandhabt werden können.
- OSGi Services werden als POJOs implementiert. Es müssen keine frameworkspezifischen Schnittstellen implementiert werden.
- OSGi ist sehr populär und weit verbreitet und findet daher in unterschiedlichen Marktsegmenten Anwendung.
- Das Framework ist relativ kompakt und überschaubar. Das bedeutet, dass die Technologie entsprechend schnell erlernbar ist.

Nachteile

- Der Entwickler muss es verstehen, Softwareanwendungen in Module zu unterteilen.
- Mit OSGi ist es nur möglich, Softwaremodule und Services innerhalb einer JAVA Virtual Machine zu verwalten. OSGi ist nicht für verteilte Anwendungen konzipiert.

3.2 Class Loading Mechanismen

Im Unterschied zu JAVA Anwendungen, die nicht auf der OSGi Service Platform aufsetzen und ein hierarchisches Class Loading System verwenden, macht OSGi von einem graphenähnlichen Class Loading System Gebrauch. Daraus resultiert der Vorteil von OSGi, Softwaremodule flexibel und dynamisch zu verwalten.

3.3 Potential von OSGi in JAVA Enterprise Edition

Application Server wie JBoss 5.0 und GlassFish Prelude V3 sind zwar selbst auf OSGi aufgebaut, unterstützen den JEE Entwickler allerdings nicht durch OSGi Features. Die Ent-

wicklung von JEE Anwendung erfolgt ausschließlich auf Basis der JEE Spezifikation, die keine OSGi Elemente inkludiert.

3.3.1 Spring-DM

Spring-DM integriert das Spring Framework in die OSGi Service Platform. Spring Anwendungen werden damit modularisierbar, versionierbar und können dynamisch verwaltet werden.

3.3.2 SpringSource dm Application Server

Eine sehr innovative Lösung für einen Application Server bietet der SpringSource dm Application Server. Der dm Server verbindet das Spring Framework und OSGi über den Spring-DM-Kernel. OSGi Features werden daher auch für den JEE Entwickler bereitgestellt. Die Entwicklung von Bundles für den dm Server ähnelt der von Spring-DM-Bundles.

Der dm Server bietet dem JEE Entwickler die folgenden Features:

- Serverseitige Anwendungen werden als OSGi Bundles implementiert und somit modularisiert.
- OSGi Bundles können versioniert werden.
- Serverseitige Anwendungen können zur Laufzeit des dm Servers entweder gänzlich, oder jedes Modul für sich, installiert, geändert und deinstalliert werden.

Auf Basis des JEE Standards werden vom dm Server neben Standard WAR mehrere proprietäre Deployment-Paketformen (WAR shared library, WAR shared service, Web Modules und PAR) unterstützt, welche durch die Fusionierung mit OSGi entstehen.

3.4 Newton

- Newton dient als Framework für hochdynamische, komponentenbasierte, serviceorientierte, verteilte Softwaresysteme.
- Komponenten werden in Newton als POJOs entwicklt.
- Das Newton Framework ist an mehreren Stellen erweiterbar, wodurch das Framework an spezielle Bedürfnisse, z.B. bei der Art wie Kompositionen instantiiert werden, angepasst werden kann.
- Newton Systeme können manuell von einem Administrator vorgegeben werden. Es ist allerdings auch möglich, automatisch skalierbare Systeme zu erstellen, bei denen z.B. die Anzahl der Kompositionen mit der Anzahl der Newton Instanzen skaliert.
- Die Komplexität von Newton liegt in der Konfiguration des Systems, d.h. in der Frage wie, wann und wo welche Komposition in der Newton Fabric installiert wird. Der Entwickler bzw. Administrator benötigt daher grundlegende Kenntnisse, wie verteilte Systeme optimal konzipiert werden.

4 Diskussion

4.1 OSGi

Das Ziel von OSGi ist es, eine universell verwendbare Plattform für JAVA Anwendungen zu bieten. In vielen Marktbereichen ist OSGi bereits vertreten. OSGi selbst unterstützt noch keine verteilten Systeme, obwohl durchaus Potentiale vorhanden wären, in denen OSGi zu Verbesserungen oder Vereinfachungen führen könnte.

In JSR 277 (siehe [39]) wurde bereits ein Modularisierungssystem, das in JAVA integriert ist, angefordert. Damit sollte es möglich sein, modulare und versionierte, jedoch keine dynamischen JAVA-Anwendungen realisieren zu können. Die Entwicklung dieser Anforderung wurde zwar eingestellt, dennoch besteht ein Trend dahingehend, modulare Softwareentwicklung in JAVA zu unterstützen.

An Beispielen wie Spring-DM, dm Server und Newton lässt sich erkennen, wie OSGi Features einige systemtechnische und entwicklungstechnische Vorteile für verteilte oder serverseitige Anwendungen mit sich bringen.

JEE Anwendungen setzen noch nicht auf OSGi auf, obwohl OSGi einige Vorteile liefern könnte. Einer der Gründe ist, dass die Entwicklungsprozesse von Standards relativ zeitaufwändig sind, in Relation zu den sich schnell weiterentwickelnden nicht-standardisierten Technologie-Diversitäten (wie z.B. dem dm Server).

4.2 Spring

Beim dm Server handelt es sich um ein sehr innovatives Produkt, das allerdings aufgrund seiner Neuartigkeit (erstes Release: Oktober 2008) einen noch relativ kleinen Kundenstamm hat.

Obwohl es möglich ist, diese Standard WAR's zu installieren, wird dringend darauf hingewiesen, auf die neuen Deployment-Formate (Web Modules, PAR) zu migrieren, da nur so alle Vorteile des dm Servers ausgeschöpft werden können. Allerdings entfernt man sich durch die Migration sukzessive vom JEE Standard.

Die Konzepte von Spring-DM und dm Server unterscheiden sich nur geringfügig voneinander. Der Hauptunterschied besteht darin, dass mit Spring-DM Spring-Anwendungen in einem OSGi Framework entwickelt werden. Der dm Server unterstützt dagegen JEE Anwendungen. Diese sollten dann nach und nach in einfachen Schritten zu Anwendungen migriert werden, die den Spring-DM-Anwendungen sehr ähnlich sind.

Zu den Schwächen des dm Servers zählen unter anderem, dass keine Bundle Fragments, die im OSGi Standard enthalten sind, unterstützt werden. Weiters erreicht man zwar durch die Bündelung von Packages zu Libraries, dass die Import-Package Anweisungen übersichtlicher werden, allerdings liegt hier ein ähnliches Problem wie bei der Verwendung von Bundle-Import vor. Es werden oft mehr Packages importiert als tatsächlich benötigt werden. Die Forderung der wohldefinierten Schnittstelle wird damit unterwandert.

Produkte wie der SpringSource dm Application Server lassen einen Trend für zukünftige Entwicklungen im Bereich der serverseitigen Anwendungen erkennen: Sie bewegen sich weg von monolithischen und statischen, hin zu modularen und dynamischen Anwendungen.

4.3 Newton

Newton macht sich die Technologie von OSGi zunutze und stellt damit ein hochdynamisches, erweiterbares Framework für verteilte Systeme dar.

Newton unterstützt ein komponentenbasiertes System, das aber wesentlich flexiblere Komponentenstrukturen zulässt als vergleichsweise JEE Applikationen. JEE Komponenten sind in den diversen Spezifikationen festgelegt und müssen sich an diese Vorgaben halten, während die Komponenten für SCA Kompositionen an keine Spezifikation gebunden sind.

Durch die gebotene Dynamik des Frameworks und die flexiblen Komponentenstrukturen, aber auch durch die manuelle und automatische Systemstrukturverwaltung (mittels Newton System Manager bzw. Provisioning Serivce), bietet Newton das Potential für eine Cloud Computing bzw. Grid Computing Plattform (siehe [6]).

5 Literatur

1 Parnas D. L: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 5(12):1053–1058, Dezember 1972. Erneut gedruckt in [2].

2 Yourdon E. N: *Classics in Software Engineering*. Yourdon Press; 1979.

3 Müller P: *Modular Specification and Verification of Object-Oriented Software*; 1.4 Modularity Aspects, Specifications, and Proofs; Berlin, Heidelberg: Springer; 2002.

4 *JAVA Archive (JAR) Features*. Enthalten in [5].

5 *JAVA 2 SDK Documentation*. Version 1.4.2. http://JAVA.sun.com/j2se/1.4.2/docs/

6 Schiller A, Springer T: *Verteilte Systeme*. Springer Berlin Heidelberg New York; 2007.

7 OSGi Alliance. OSGi™ – The dynamic module system for JAVA™ ; 2009. http://www.osgi.org.

8 *JAVA Naming and Directory Interface*. Version 1.4.1. http://JAVA.sun.com/j2se/1.4.2/docs/guide/jndi/index.html. Enthalten in [5].

9 Wütherich G, Hartmann N, et al: *Die OSGi Service Platform*. Dpunkt.verlag Heidelberg; 2008.

10 Bartlett N: *OSGi in Practice*. Creative Commons. Derzeit als Entwurf unter http://neilbartlett.name/blog/osgibook/ einsehbar; 2009.

11 *OSGi Service Platform Core Specification*. Release 4. Version 4.1; April 2007. Erhältlich unter [7].

12 *OSGi Service Platform Service Compendium*. Release 4. Version 4.1; April 2007. Erhältlich unter [7].

13 Howes H: *RFC 1960 – A String Representation of LDAP Search Filters*. http://www.ietf.org/rfc/rfc1960.txt; Juni 1996.

14 Ullenboom C: *JAVA ist auch eine Insel - Programmieren mit der JAVA Standard Edition Version 6*. Galileo Computing, 7. Auflage. ISBN 978-3-8362-1146-8. 2007

15 Lahres B, Rayman G: *Praxisbuch Objektorientierung – Professionelle Entwurfsverfahren*. Galileo Computing. ISBN 3-89842-624-6. August 2006.

16 Gamma E, Helm R, et al: *Design Patterns*. Addison-Wesley. 1995.

17 *JAVA Plattform Enterprise Edition (JAVA EE) Specification.* v5. Sun Microsystems, Inc. 2006. Erhältlich unter [18].

18 Developer Resouce for JAVA Technology. JAVA Enterprise Edition. http://JAVA.sun.com/JAVAee/. 2009.

19 Community driven open source middleware. JBoss Community. www.jboss.org. 2009.

20 *Sun GlassFish Enterprise Server v3 Prelude Add-On Component Development Guide;* 2008. Erhältlich unter [21].

21 GlassFish Community. https://glassfish.dev.JAVA.net/. 2009.

22 Johnson R: *J2EE Development without EJB*. Wiley Publishing, Inc. Indianapolis, Indiana; 2002.

23 Spring Source Community. http://www.springsource.org/. 2009.

24 *Spring Framework – JAVA/J2EE Application Framework – Reference Document*. Version 2.5.6

25 JAVA Platform, Enterprise Edition. Enterprise JAVABeans Technology. http://JAVA.sun.com/products/ejb/. 2009.

26 Rupp H: *EJB für Umsteiger*. 1. Auflage. dpunkt.verlag, Heidelberg. 2007.

27 Colyer A, Hildebrand H, et. al: *Spring Dynamic Modules Reference Guide*. Version 1.1.2. 2008.

28 Spring Source Community. Spring Dynamic Modules for OSGi(tm) Service Platform. www.springsource.org/osgi. 2009.

29 Spring Source Community. SpringSource dm Server. www.springsource.org/dmserver. 2009.

30 Laddad R, Yates C, et. al: *SpringSource dm Server Programmers Guide*. Version 1.0.1. 2008.

31 Harrop R, Kuzan P, et al: *SpringSource dm Server User Guide*. Version 1.0.1. 2008.

32 *JAVA Servlet Specification*. Version 2.5 MR6. 2007. Erhältlich unter [18].

33 The Community Resource for Jini Technology. Jini.org. http://www.jini.org/wiki/Main_Page. 2009.

34 SCA Assembly Model Specification. Version 1.0.0. 2007.

35 Open SOA. Service Component Architecture Home. http://www.osoa.org/display/Main/Service+Component+Architecture+Home. 2009.

36 Home – Open Service Oriented Architecture. http://www.osoa.org/display/Main/Home. 2009.

37 *Newton 1.4-b52: Developer Guide*. 2009. Erhältlich unter [38].

38 Newton Framework. Code Cauldron. http://newton.codecauldron.org/site/index.html. 2009.

39 JAVA Specification Requests. *JSR-277: JAVA Module System*. The JAVA Community Process Program. http://jcp.org/en/jsr/detail?id=277. 2009.