Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Steffen Kuche

Entwurf und Realisierung einer Content Management System Komponente auf Basis der OSGi Service Plattform.

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Steffen Kuche

Entwurf und Realisierung einer Content Management System Komponente auf Basis der OSGi Service Plattform.

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. rer. nat. Bernd Kahlbrandt

Abgegeben am 09.02.2010

Steffen Kuche

Thema der Bachelorarbeit
Entwurf und Realisierung einer Content Management System Komponente auf Basis der OSGi Service Plattform.
Stichworte
OSGi, Content Management Systeme, Apache Felix, Java Content Repository
Kurzzusammenfassung
In dieser Arbeit wird untersucht, inwiefern sich die OSGi Service Plattform dazu eignet, Content Management System Komponenten zu entwickeln und zu betreiben. Unter anderem werden dabei die Grundlagen für OSGi, Content Management Systemen und für Java Content Repositorys untersucht. Des Weiteren werden verschiedene Ansätze überprüft, die die Entwicklung von Applikationen auf Basis der OSGi Service Plattform vereinfachen. Zu der Entwicklung der CMS Komponente gehört ein OSGi spezifischer Test auf Basis eines daraufhin ausgelegten Testframeworks. Außerdem wird das Konfigurationsmanagement eines OSGi Projekts untersucht.

Steffen Kuche

Title of the paper
Design and realization of a Content Management System component based on the OSGi Service Platform.
Keywords
OSGi, Content Management Systems, Apache Felix, Java Content Repository
Abstract
This study examines if it is appropriate to develop a Content Management System component based on the OSGi Service Platform. The basics of OSGi, Content Management Systems and Java Content Repositories are investigated. Furthermore are different approaches considered to simplify the development of applications based on the OSGi Service Platform. The development of the CMS component includes an OSGi specific test. In addition the configuration management of an OSGi project is examined.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziele dieser Bachelorarbeit

In dieser Bachelorarbeit wird auf Basis der OSGi Service Plattform eine Komponente eines Content Management System entwickelt, um zu untersuchen wie sinnvoll sich diese beiden Technologien verbinden lassen. Weiterhin werden verschiedene Konzepte untersucht, die die Entwicklung von OSGi Applikation vereinfachen und einander gegenübergestellt. Es werden Content Management Systeme betrachtet, welche als Persistenzschicht ein Java Content Repository (JSR 170 und JSR 283) nutzen. Als OSGi Service Plattform wird Apache Felix zum Einsatz kommen.
Um die Funktionalität der Komponente zu veranschaulichen wird ein explorativer Prototyp entwickelt. Dieser beinhaltet sowohl die Erfassungs- als auch die Ausspielungskomponente eines CMS. Die Ausspielungskomponente ist für das World Wide Web konzipiert und unter Verwendung von Apache Sling realisiert. Es wird das Konfigurationsmanagement eines solchen Projekts betrachtet und eine Teststrategie entwickelt.

1.2 Motivation

In vielen Softwareprojekten muss mit einer hohen Komplexität umgegangen werden. Bedingt durch den hohen Verbreitungsgrad der Programmiersprache Java entstehen immer komplexere Java basierte Anwendungen. Ein weit verbreitetes Mittel um Komplexität zu handhaben ist das Mittel der Modularisierung. In Java gestaltet sich die Umsetzung dieses Konzepts aber außerordentlich schwierig. Eine entsprechende Sprachunterstützung fehlt. Als Konsequenz entstehen monolithische Anwendungen, die dynamischen Anforderungen nicht gerecht werden. Außerdem gestaltet es sich so schwierig, die Entwicklungsarbeit auf unabhängige Teams aufzuteilen.
Entstehen neue Anforderungen oder soll die Anwendung an eine andere Umgebung ange-

passt werden, müssen Bestandteile häufig neu entwickelt werden. Dies ist einer zu hohen Kopplung auf Klassenebene geschuldet. Eine Wiederverwendung von bestehenden Komponenten scheitert schon daran, dass es nicht möglich ist durch die Sprache Java eine Komponente zu definieren und zu versionieren. OSGi ist eine Antwort auf diese Defizite von Java. Es stellt ein dynamisches Modulsystem für Java bereit. Dabei hat sich OSGi in den letzten Jahren von seiner ursprünglichen Anwendung in eingebetteten Systemen emanzipiert und wird mittlerweile in den unterschiedlichsten Bereichen verwendet. Von Anwendungen für Mobilfunkgeräte über Client-Anwendungen wie der Eclipse IDE und ihrer RCP Architektur bis hin zu Server- und Webapplikationen.
Content Management Systeme erfüllen immer mehr Aufgaben und sind somit auch einer steigenden Komplexität ausgesetzt. Weiterhin sind Content Management Systeme häufig ein integraler Bestandteil der IT-Infrastruktur. Folglich müssen diese an die jeweilige Umgebung angepasst werden. Dies kann z.B. die Nutzung von bestehenden Authentifizierungs-Mechanismen sein, oder aber eine spezifisches Verhalten bei Statusübergängen von Dokumenten. Um solche Anforderungen umzusetzen, bietet sich ein dynamisches Konzept wie OSGi an.

Diese Arbeit ist in Abstimmung mit dem Unternehmen subshell entstanden. Subshell ist der Herausgeber des CMS Sophora, auf welches in der Arbeit mehrfach Bezug genommen wird.

1.3 Konkretisierte Aufgabenstellung

Die vorliegende Arbeit soll den Einsatz der Technologie OSGi im Rahmen eines Content Management Systems beleuchten.
Zu diesem Zweck werden zunächst Content Management Systeme an sich betrachtet um konkrete Einsatzgebiete von OSGi in diesem Kontext zu erörtern. Weiterhin wird untersucht welchen Einfluss die Verwendung von OSGi auf den gesamten Software-Lebenszyklus hat, einschließlich der Entwicklung und dem Betrieb.
Folgende Meilensteine ergeben sich vor diesem Hintergrund:

1. Vorstellung von Content Management Systemen, deren Einsatzgebiete und deren technische Architektur. Erörterung von elementaren technischen Voraussetzung von CMS.

2. Erarbeitung der Grundlagen von OSGi, eine Einordnung dieser Technologie und die Darstellung der Probleme zu dessen Lösung OSGi konzipiert wurde. Ferner soll eine Implementierung dieser Spezifikation vorgestellt werden. Weiterhin ist ein Ausblick in die Ambitionen von Java nötig, Konzepte der Modularisierung in den Sprachumfang aufzunehmen.

3. In dem Umfeld von OSGi sind mehrere Frameworks entstanden, die es ermöglichen dynamische Services deklarative zu handhaben. Diese werden untersucht, einander gegenübergestellt und entschieden welches dieser Frameworks im Rahmen dieser Arbeit verwendet wird.

4. Es muss ein konkretes Anwendungsgebiet identifiziert und analysiert werden.

5. Ein explorativer Prototyp eines CMS wird benötigt. Dieser wird zunächst entworfen und anschließend implementiert.

6. Die Anwendung muss entworfen und implementiert werden. Hierzu gehört die Betrachtung des Build-Prozesses, die Integration von OSGi in die Entwicklungsumgebung und die Konzeption von OSGi spezifischen Testfällen.

1.4 Konventionen

Im Text werden häufig englische Begriffe verwendet, dies liegt daran, dass diese in der Informatik unumgänglich sind und oftmals keine gängigen deutsche Übersetzung existieren. Ihre deutsche Übersetzung würde ein erschwertes Verständnis zur Folge haben. Um an dieser Stelle Verwirrung zu vermeiden, werden durchgängig die englischen Begriffe, als Eigennamen ohne entsprechende Kennzeichnung, verwendet. Wobei auch darauf geachtet wurde keine unnötigen Anglizismen zu verwenden und insbesondere das so genannte „Denglisch“ wie geupdatet, zu vermeiden. Aus dem englischen entlehnte Wörter sind durchgängig klein geschrieben, es sei denn es handelt sich um einen festen Begriff wie z.B. „Content Management System“. Wenn im Text auf Quellcode oder CND - Fragmente Bezug genommen wird, sind diese *kursiv* gekennzeichnet. Verzeichnisse Dateinamen oder URLs werden in einer `Schreibmaschinen - Schrift` dargestellt.

Die dargestellten Code - Fragmente konzentrieren sich auf das Wesentliche und erheben keinerlei Anspruch auf Vollständigkeit.

1.5 Gliederung der Arbeit

Diese Arbeit besteht im Wesentlichen aus acht Kapiteln, die im Folgenden kurz dargestellt werden.

Im zweiten Kapitel werden zunächst die relevanten Grundlagen erarbeitet und veranschaulicht. Dazu gehören zunächst Content Management Systeme an sich.
Wenn man sich mit modernen CMS beschäftigt, stößt man zwangsläufig auf die Technologie

der Java Content Repositories. Diese werden in vielen modernen CMS zur Verwaltung des Contents verwendet. Das Konzept, die Verwendung und deren Einsatz wird erläutert. Der dritte Abschnitt dieses Kapitels widmet sich der Modularität. In dem anschließenden vierten Abschnitt wird OSGi behandelt. Unter anderem wird in diesem Abschnitt auf die Spezifikation von OSGi eingegangen. Eine Implementierung von OSGi, Apache Felix wird vorgestellt. Außerdem wird Ap,

ache Sling vorgestellt. Dies ist ein Web-Framework, das auf einem Java Content Repository basiert. Es wird im Rahmen des explorativen Prototyps verwendet.

Kapitel drei widmet sich der Vorstellung von verschiedenen Ansätzen zum Umgang mit dynamischen Services. Es wird motiviert, warum ein zusätzliches Framework nützlich ist und wie die OSGi Spezifikation mit den angesprochen Problemen umgeht.

Das vierte Kapitel Analyse motiviert die fachliche Problemstellung. In einem Anwendungsszenario wird die praktische Relevanz der in den folgenden Kapiteln entworfenen Komponente erörtert. Dazu gehören Anwendungsfälle und ein explorativer Prototyp.

Kapitel fünf Entwurf beinhaltet den technischen Entwurf der Komponente. Weiterhin wird die Komponente in Module unterteilt und deren Funktionalität und Verantwortlichkeit innerhalb der Komponente dargestellt. Die Interaktion der Module zur Laufzeit wird hier veranschaulicht.

Kapitel sechs widmet sich der Implementation der Komponente. Dabei wird sowohl auf Implementierungsdetails als auch auf das Konfigurationsmanagement eingegangen.

Das siebte Kapitel behandelt Testfälle auf Basis der in Kapitel vier entworfenen Anwendungsfälle.

In dem achten Kapitel werden die Ergebnisse dieser Arbeit bewertet. Darüber hinaus wird hier Kritik geübt und es wird betrachtet welche Bedeutung die Ambitionen von Java zur Integration eines Konzepts der Modularisierung für OSGi hat.

In dem abschließenden Kapitel werden die Ergebnisse zusammengestellt und ein Ausblick gegeben.

Kapitel 2

Grundlagen

2.1 Content Management

Um Content Management genauer zu betrachtet ist es zunächst notwendig die Bedeutung des Begriffs Content zu klären.

2.1.1 Content

Der Begriff Content ist sehr weit gefasst und unspezifisch. Dies charakterisiert aber auch die Art und Weise, wie dieser Begriff verwendet wird. Bereits 1998 wurde eine Arbeitsgruppe von der „Society of Motion Picture and Televsion Engineers“ und der „Europen Broadcasting Union“ gegründet, um unter anderem diesen Begriff näher zu spezifizieren. Nach dieser Arbeitsgruppe wird Content durch essence und metadata definiert. Essence ist in diesem Zusammenhang der eigentliche Inhalt, der in beliebiger Form vorliegen kann. Dies sind z.B. einfache Texte, Bilder, Videos, Referenzen zu anderen Dokumenten, Gliederungen und dergleichen mehr. Metadata hingegen sind Daten, welche die essence näher beschreiben. Dies kann bei einem Photo die Auflösung, seine geographischen Koordinaten, die Lizenz unter der es zur Verfügung steht oder z.B. Relationen zu anderen Photos sein (vgl. (SMP98), S. 63 ff.).
In Abbildung 2.1 wird diese Strukturierung dargestellt.

Abgrenzung zwischen Daten, Informationen und Content Daten sind die kleinsten Informationseinheiten in der Informatik. Diese können Zahlen, Wörter, Buchstaben, Audio oder Video Dateien sein. Auf einem niedrigen Abstraktionsniveau wird in einem Computer alles als Datum angesehen, gespeichert, verarbeitet und ausgegeben. Dies spiegelt sich auch in dem Eingabe-Verarbeitung-Ausgabe (EVA) Prinzip wider.
Informationen hingegen sind keine kleinen Dateneinheiten, sondern vielmehr eine Antwort auf eine gewisse Fragestellung. Es sind gruppierte Daten oder Datenobjekte, die Bedeutung,

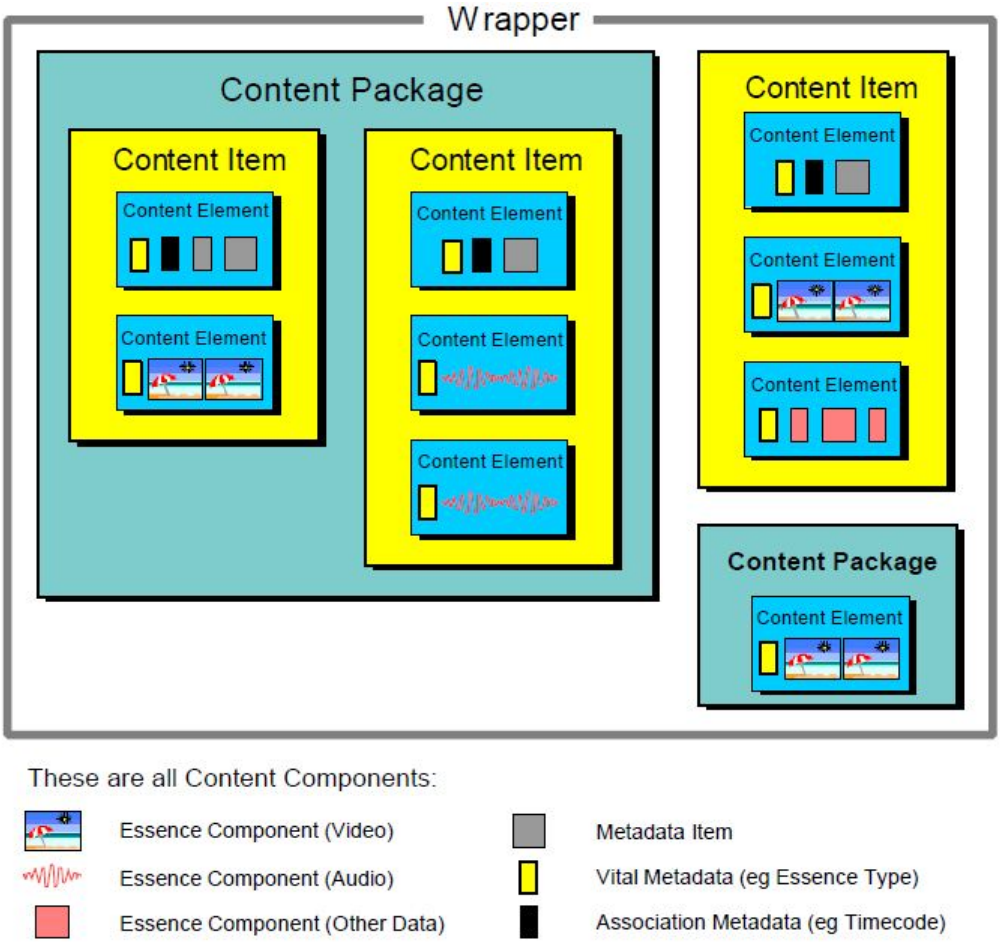

Abbildung 2.1: Struktur von Content ((SMP98), S. 67).

Relevanz und Nutzen haben. In ihnen ist bereits eine Interpretation enthalten. Sie haben eine inhärente Semantik, welche als solche nicht von Computern verarbeitet werden kann.

> „Information is what human beings transform their knowledge into when they want to communicate it to other people. It is knowledge made visible or audible, in written or printed words or in speech.“ ((Orn04), S. 7)

Daten sind immer bloß Daten als solche. Bei Informationen verhält es sich anders. Sie sind nicht nur das, was sie ausdrücken. sondern entfalten ihre „Informationen“ erst in der Interpretation und dem Kontext; diese sind dem Computer aber unzugänglich.
Der Content behebt diese Unzulänglichkeit durch die Metadaten. Die Metadaten beschreiben die Informationen durch Daten, in solch einer Weise, dass sie für die maschinelle Verarbeitung zugänglich gemacht werden. So ist Content nichts anderes, als durch Metadaten angereicherte Informationen (vgl. (Boi05), S. 3 ff).

2.1.2 Content Management System

Ein Content Management System (CMS) unterstützt seine Anwender in erster Linie in folgenden Anwendungsfällen(vgl. (Boi05),S .111 ff.; (Fer07), S. 53 ff.):

- Erstellen von Content
- Management und Organisation von Content
- Modifikation von Content
- Präsentation und Publikation von Content

Diese Kernkompetenzen haben alle CMS gemeinsam, allerdings hängt der Funktionsumfang stark von dem jeweiligen Anwendungskontext und dem zu verwaltenden Content ab. Bob Boiko beschreibt dies in seinem Buch „Content Management Bible 2nd Edition“ wie folgt:

> „Like most things that have complexity, Content Management means different things to different people.“ ((Boi05), S. 65)

Des Weiteren existieren eine Reihe von Technologien, die mit CMS verwandt und deren Grenzen fließend sind. Diese sind z.B.:

- Digital Asset Management (DAM)
- Document Management System (DMS)
- Enterprise Content Management (ECM)
- Web Content Management System (WCMS)

Aus diesen Gründen beschränke ich die Betrachtung von Content Management Systemen auf solche, welche in Redaktionen eingesetzt werden und hauptsächlich dazu dienen Content zu produzieren, zu organisieren und zu publizieren. Die Puplikationsform beschränkt sich hier auf das World Wide Web. Ein solches CMS muss den Produktionsprozess unterstützen, den Arbeitsprozess der Redaktion abbilden können und Funktionalitäten anbieten um gemeinschaftlich und verteilt Content zu produzieren. Daraus ergeben sich eine weitere Anzahl von Anforderungen wie z.B. die Steuerung der Zugriffsrechte, eine Versionierung und ein Statuskonzept für die Inhalte. Alle Inhalte befinden sich so immer in einem definierten Zustand, wie z.B. archiviert oder veröffentlicht. Des Weiteren wird durch ein CMS der Content von dem Layout und die Content Produzierenden von den technischen „Hindernissen“ getrennt. So ist es möglich, dass von Redakteuren erstellter Content in unterschiedlichen Publikationskanälen wie WAP, WWW oder Print - Medien veröffentlicht wird, ohne dass diese technisches Know-How oder gestalterische Fähigkeiten besitzen müssen.
Weiterhin heißt es in (Boi05), S66:

> „From a business goals perspective, CM distributes business value. From an analysis perspective, CM balances organizational forces. From a professional perspective, CM combines content-related disciplines. From a process perspective, CM collects, manages, and publishes information. From a technical perspective, CM is a technical infrastructure."

Abgrezung CM und CMS Content Management (CM) wird von Content Management System unterschieden. Content Management beschreibt einen Prozess Publikationen zu organisieren. Er beinhaltet die Ablauforganisation, gemeinschaftlich und verteilt Content zu produzieren und zu publizieren. Im weiteren Sinne ist CM eine Zusammenfassung von Geschäftsprozessen, mit deren Hilfe Unternehmen einen großen Bestand an Informationen effektiv handhaben.
Hingegen ist ein CMS ein fertiges Stück Software. Es setzt die abstrakten CM Anforderungen programmatisch um und löst sie somit (vgl. (RR02), S. 15 - 16; (Boi05), S. 65 ff.).

2.1.2.1 Komponenten eines Content Managements Systems

Jedes CMS weist seine eigene Architektur auf. Diese ist stark vom jeweiligen Anwendungskontext abhängig. Dennoch kann man drei Komponenten eines CMS als deren Hauptbestandteil ansehen, die Erfassungs-, die Verwaltungs- und die Ausspielungskomponente. Abbildung 2.2 zeigt eine schematische Übersicht dieser Komponenten.

Erfassungskomponente Die Erfassungskomponente umfasst alles um Content zu produzieren, zu erfassen, zu konvertieren und zu aggregieren. Alles was durch das CMS verwaltet und publiziert werden soll, muss durch die Erfassungskomponente erfasst und organisiert werden. Hierbei kommen mehrere Quellen in Frage. Die wichtigste Quelle ist das Erfassen, also das Produzieren von Content. Die Erfassungskomponente muss die hierfür nötige Unterstützung bieten, wie Hilfestellung durch eine zweckmäßige Usability, dem Abbilden des Produktionsprozesses, eine Versionskontrolle oder z.B. dem Bereitstellen von Templates. Weiterhin können bestimmte Quellen automatisch importiert werden, dies kann z.B. ein RSS Feed, ein Photo- oder Video-Archiv sein. Der so erfasste Content muss in einem Format vorliegen, das eine weitere Verarbeitung erlaubt. Hier kommt die Konvertierung ins Spiel. Diese ist in der Lage aus einer Reihe inkompatibler Formate kompatible zu erzeugen und unerwünschte Informationen zu entfernen. Durch die Zusammenstellung von Content, der Aggregation, können Themengebiete vernetzt oder multimedial aufbereitet werden.

Verwaltungskomponente Innerhalb der Verwaltungskomponente ist die Persistenzschicht angesiedelt. Sowohl die Administration des CMS als auch die Anbindung an andere Systeme wie z.B. einer Suchmaschine, erfolgt hier. Weiterhin können hier Konfigurationen für die

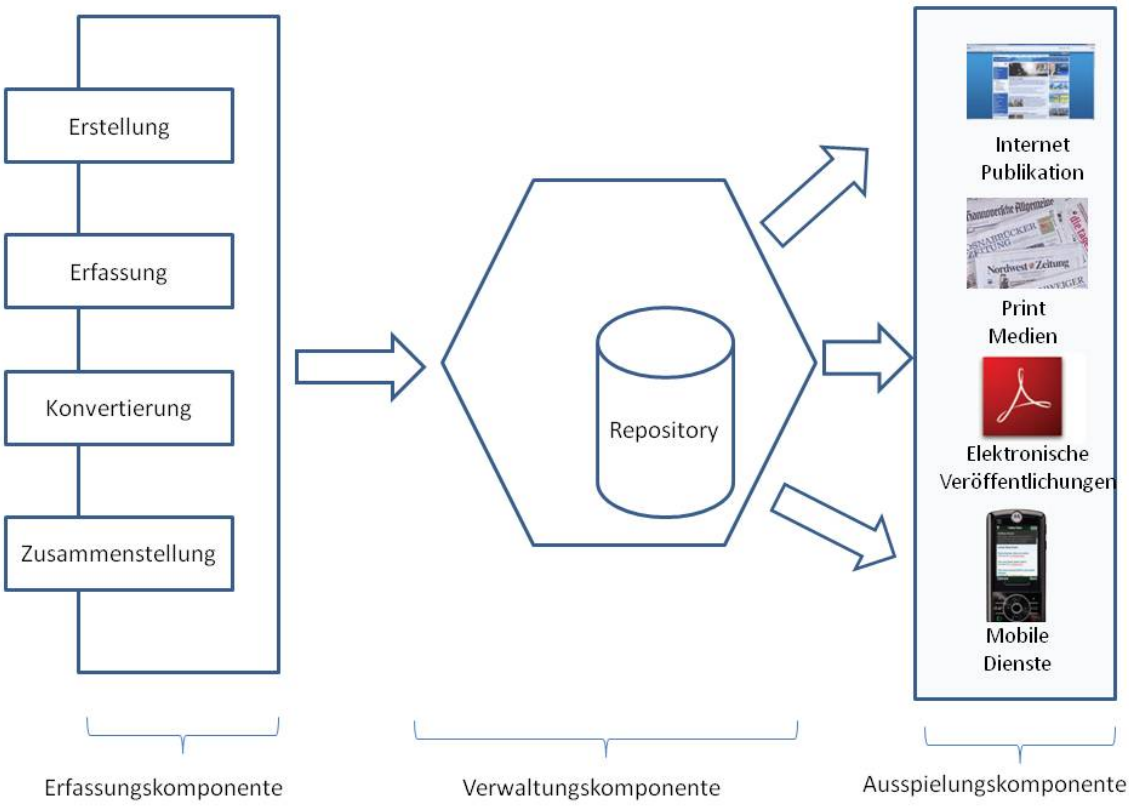

Abbildung 2.2: Überblick der Komponenten eines CMS. Angelehnt an ((Boi05), S. 86).

Erfassungs - und die Ausspielungskomponente hinterlegt und bearbeitet werden. Die Benutzerverwaltung ist in dieser Komponente zu finden und es wird eine Schnittstelle für die Ausspielungskomponente angeboten. Zusätzlich werden die Templates der Erfassungskomponente hier an zentraler Stelle abgelegt. Auch werden Cronjobs, regelmäßig ablaufende Routinen, sowie Regeln für den Arbeitsablauf in dieser Komponente verwaltet. In der Persistenzschicht wird der gesamte Content, mit der Versionierung gespeichert.

Ausspielungskomponente Die Ausspielungskomponente ist dafür verantwortlich Daten aus der Persistenzschicht auszulesen um daraus automatisiert Publikationen zu erstellen. Die Art der Publikation ist hierbei zunächst irrelevant. Die Erstellung der Publikationen erfolgt durch Templates. In diesen Templates wird der Content aus der Persistenzschicht mit Layout angereichert und für die gewünschte Puplikationsform aufbereitet. Diese Templates bieten mehr Funktionalität an als jene, welche man in der Erfassungskomponente antrifft.

Vielmehr wird von diesen Templates eine Programmiersprache wie JSP, PHP oder XSL - FO verwendet, um den Content in die gewünschte Form zu transformieren.

2.2 Java Content Repository

Die Technologie der Java Content Repository gab es bereits vor einer einheitlichen Spezifikation durch den Java Specification Request 170 im Jahr 2002. Dieser hat aber deren Entwicklung gefördert und zur Verbreitung der Technologie beigetragen. Seit 2002 wird die Spezifikation unter dem Java Community Process weiterentwickelt. Die Entwicklung der Spezifikation wird zu einem bedeutenden Anteil von Mitarbeitern der Day Software, Inc. vorangetrieben. Die finale Version der Spezifikation wurde am 17.06.2005 verabschiedet (vgl. (JSR)). Im September 2005 wurde der Java Specification Request 283 ins Leben gerufen um die Entwicklung der JSR API Version 2.0 zu beschreiben. Es existieren zahlreiche Implementierungen der Spezifikation, wobei hier nur die Implementierung Apache Jackrabbit behandelt wird. Diese Implementierung ist am weitesten verbreitet und unter der Apache License frei verfügbar. Unter den unterschiedlichen Implementierung sei hier nur noch auf Priha[1] und das Alfresco Java Content Repository[2] hingewiesen. Priha ist eine sehr schlanke Implementierung und wird bevorzugt als eingebettetes JCR verwendet.

2.2.1 Funktionen eines Java Content Repositories

Ein Java Content Repository bietet eine einheitliche Schnittstelle an um Daten zu speichern und zu lesen. Es ist ein Informations Management System und bietet eine Vielzahl von Services an um Daten zu speichern, abzurufen und zu verwalten. Dabei werden die Stärken von relationalen Datenbanken, wie referenzielle Integrität, Transaktionen oder das sichere Speichern von strukturierten Daten um weitere Funktionen erweitert. Dazu gehört eine Versionierung, Daten Observation, Vererbung und ein fein granulares Sicherheitskonzept. Weiterhin können unstrukturierte und hierarchische Daten gehandhabt werden. Es werden SQL Abfragen und eine Teilmenge von XPath zum Selektieren von Daten unterstützt. Weiterhin existieren Export und Import Schnittstellen. Diese arbeiten mit der Zwischenrepräsentation XML. Für die Transaktionsverwaltung wird die Java Transaction API (JTA) verwendet. Ein JCR dient dabei auch immer nur als Schnittstelle zwischen dem Anwendungskontext und der Persistenzschicht.

[1] `http://www.priha.org/`
[2] `http://www.alfresco.com/`

2.2.2 Ziele der JSR 170 und JSR 283

Als im Jahr 2002 der JSR 170 initiiert wurde, gab es bereits eine steigende Anzahl von Anbietern proprietärer Content Repositories. Ein Ziel des JSR 170 ist es, eine einheitliche Schnittstelle auf solch ein Repository zu schaffen. Dies erspart einerseits den Entwicklern das Erlernen der speziellen API, andererseits erleichtert es schließlich die Entwicklung von Content zentrierten Anwendungen, indem es von der konkreten Implementierung des JCR und der Persistenz-Schicht abstrahiert (vgl.(NP09), S. 11). Als Persistenz-Schicht können verschiedene Technologien zum Einsatz kommen, wie z.B. relationale Datenbanken, objektorientierte Datenbanken, XML Dateien oder auch ein Dateisystem. Dank der einheitlichen Schnittstelle kann nicht nur die konkrete Implementierung des Content Repositories, sondern auch die zugrunde liegende Persistenzschicht ausgetauscht werden, ohne in höheren Abstraktionsschicht, in der Anwendungslogik Änderungen vorzunehmen. Abbildung 2.3 verdeutlicht diesen Sachverhalt. Die beiden alten content repositories, zwei und drei, werden durch ein viertes ersetzt. Zu beachten ist in diesem Fall, dass sich hier auch die Persistenzschichten ändern. Trotzdem sind keine Änderungen in der Anwendungslogik zu erwarten. Die Änderungen beschränken sich im Idealfall auf eine Umkonfiguration in der Anwendung, z.B. dadurch bedingt, dass das vierte Content Repository unter einer anderen Adresse erreichbar ist. Eine Daten Migration bleibt aber unumgänglich und wird von der Spezifikation nicht mit abgedeckt. Die Möglichkeit das Content Repository zu wechseln, stellt in der Praxis aber den größeren Vorteil dar. Die Struktur der Daten ist durch ihre persistente Form definiert und darauf abgestimmt. Tauscht man z.B. ein Filesystem gegen eine Datenbank aus, wird man in der Datenbank nur den Datentyp BLOB vorfinden. Dies wird zwar durch die meisten Datenbanken unterstützt, zielt aber nicht auf die eigentliche Stärken einer solchen ab. Daten und Persistenzschicht sind interdependent voneinander abhängig und eine Auswechslung der Persistenzschicht ist nur dann sinnvoll, wenn die Daten zukünftig andersartig genutzt werden.

Aufbau der Spezifikation Der JSR 170 ist in drei separate Level aufgeteilt, um es Herstellern von JCR zu erleichtern mit ihren Produkten nur einen Teil der Spezifikation zu erfüllen. Die Level bauen aufeinander auf, d.h. es ist nicht sinnvoll Level zwei zu implementieren ohne die Funktionen aus Level eins zu berücksichtigen.

Level 1: Dieser Level spezifiziert ein JCR, welches von Anwendungen verwendet wird, die lesend auf das Repository zugreifen. Es handelt sich um eine read-only API. Diese unterstützt ausschließlich das Auslesen von nodes properties. Zusätzlich schreibt es eine Export Schnittstelle vor. (vgl. (NP09), S. 47)

Level 2: Dieser Level erweitert Level eins um Schreibende Funktionen und ermöglicht so eine bidirektionale Interaktion der Anwendung mit dem JCR. (vgl. (NP09), S. 166)

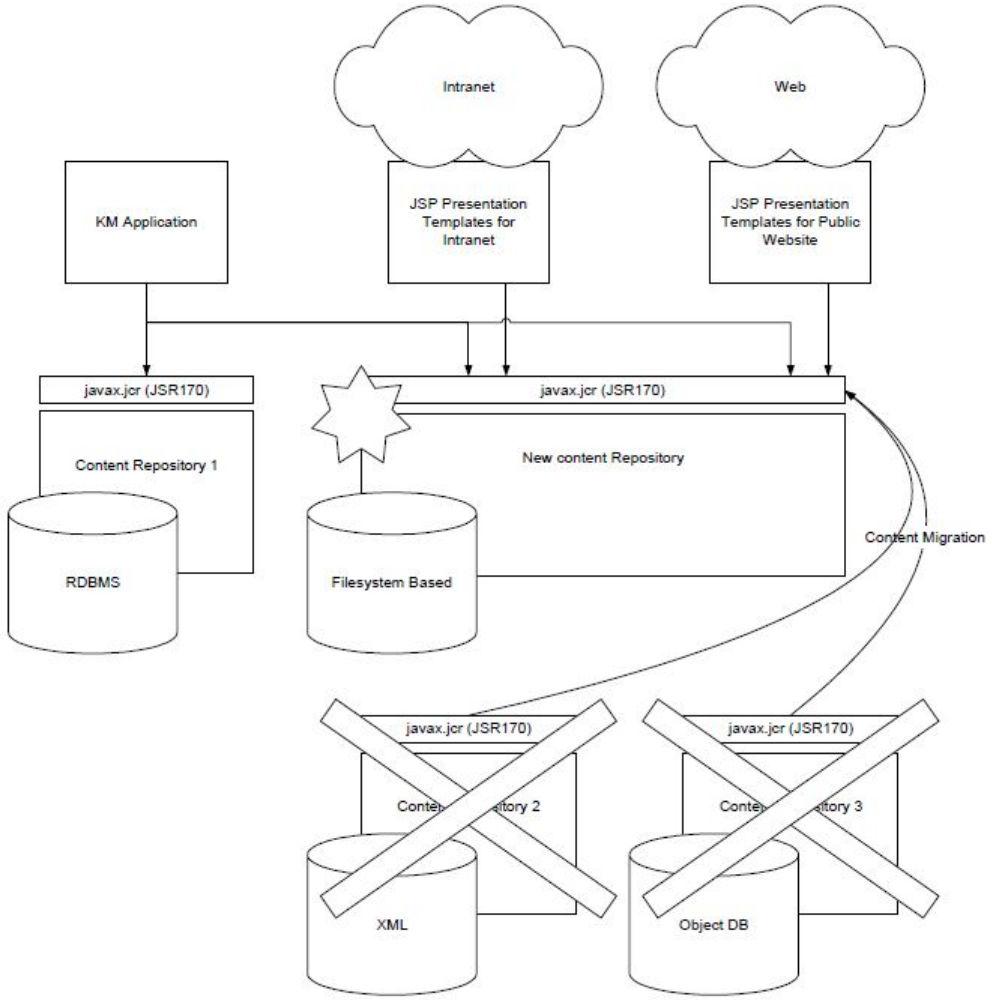

Abbildung 2.3: Einheitliche Schnittstelle für Content Repositories. (NP09), S. 14

Erweiterte Funktionalität: Dieses Level sieht funktionale Erweiterungen des JCR vor. Diese sind: (vgl. (NP09), S. 225)

- Versionierung
- Observation von Daten[3]
- Transaktionen durch JTA

[3]Hinter der Observation von Daten verbirgt sich ein anders Konzept, als jenes welches durch Trigger im Kontext von Datenbanken bekannt ist. Anwendungen können sich durch den Listener-Mechanismus an zuvor definierten Ereignissen registrieren und auf diese reagieren. Hingegen lösen Trigger Stored Procedures aus. Diese sind in proprietären Sprachen innerhalb der Datenbank gespeichert.

- Unterstützung von SQL Abfragen
- Explizites Sperren

2.2.3 Das Modell eines Repositories

Ein Content Repository besteht aus einem oder mehreren workspaces. In einem workspace werden die Daten in einem Baum vom Grad N organisiert. Dieser Baum hat genau einen Wurzelknoten. Innerhalb des Baums können nodes und properties gehalten werden. Eine node kann keine oder mehrere (n*) child nodes haben und keine oder mehrere properties. Jede node hat genau eine Eltern node. Properties treten immer als Blätter auf und haben genau eine Eltern node. Der gesamte Inhalt des workspace wird in diesem Baum abgelegt. Die properties können beliebige Datentypen enthalten. Jede node kann mit Hilfe einer Abfragesprache eindeutig selektiert werden. Dabei handelt es sich um eine Teilmenge von XPath. Abbildung 2.4 stellt den Baum eines Repositorys dar.

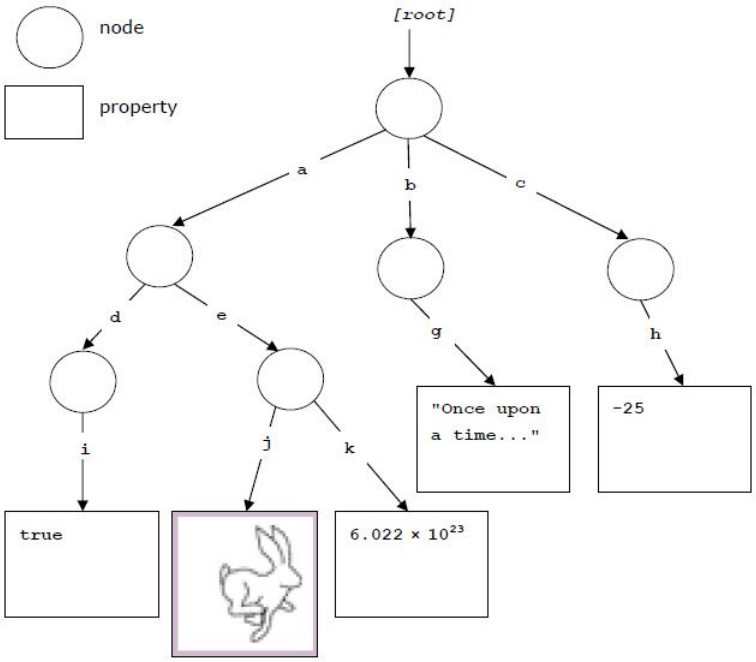

Abbildung 2.4: Das Model eines Repositories. ((NP09))

2.2.4 Apache Jackrabbit

Apache Jackrabbit ist seit März 2006 ein „top level project“ der Apache Software Foundation. Die ersten Ideen gehen auf den Februar 2002 zurück, zeitgleich mit der Initiierung des JSR 170. Es liegt momentan in der stabilen Version 1.6.0 vor und implementiert alle drei

Level der JSR 170 Spezifikation. Seit dem 26.08.2009 ist die Version 2.0 alpha9 veröffentlicht, welche die JSR 283 Spezifikation implementiert. Im Folgenden wird die Version 1.6.0 betrachtet.

Verwendung von Apache Jackrabbit

Apache Jackrabbit ist das am weitesten verbreitetste JCR und wird in vielen Content zentrierten Anwendungen verwendet. Beispielhaft sind hier die folgenden genannt:

Content Management Systeme

- Hippo [4]
- Magnolia[5]
- Sophora [6]

Enterprise Content Management

- Nuxeo [7]

Web Framework

- Apache Sling [8]

Document Management System

- OpenKM [9]

2.2.4.1 Node Types

Wie in dem Abschnitt 2.2.3 beschrieben, werden die Daten eines JCR in nodes abgelegt. Bei den nodes handelt es sich um einen flexiblen objektorientierten Ansatz. Jede node gehört zu genau einem Primary Node Type und wird durch diesen definiert. Node Types hingegen definieren sich durch ihre supernodetypes, ihren child nodes, ihren properties und ihren Mixins. Sie sind das Pendant zu den Klassen in der objektorientierten Programmierung und werden

[4] http://www.onehippo.com/
[5] http://www.magnolia-cms.com/home.html
[6] http://www.sophoracms.com/
[7] http://www.nuxeo.com/en/
[8] http://sling.apache.org/site/index.html
[9] http://www.openkm.com/

dazu verwendet domän spezifische Daten zu modellieren. Hingegen entsprechen nodes den Objekten in der objektorientierten Programmierung.
Node Types können einem Namensraum angehören. Dieser wird durch einen Doppelpunkt von dem Namen des Node Types getrennt. Dieses Konzept wurde nach dem XML - Konzept für Namensräume adaptiert und verhindert primär Namens Kollisionen (vgl. (JSR) S. 26). Die Namensräume werden in einer Registratur abgelegt. Folgenden Standard Namensraum Präfixe sind dort definiert.

- *jcr* Reserviert für Daten, welche durch vor installierte node types definiert werden.
- *nt* Reserviert für vor installierte primary node types.
- *mix* Reserviert für vor installierte Mixins.
- *xml* Aus Kompatibilitäts - Gründen zu XML reserviert.
- Der leere Präfix ist als default Namensraum reserviert.

Erweiterung von nodes durch Mixins Ein Primary Node Type kann durch Mixins erweitert werden. Mixins sind eine Gruppe von properties und oder child nodes, welche demPrimary Node Type bei Bedarf hinzugefügt werden können. Mixins haben oft eine semantische Bedeutung. Dies wird auch an ihren Bezeichnern wie etwa *mix:lockable* oder *mix:versionable,* deutlich. Ein Mixin kann im Gegensatz zu einem Primary Node Type nicht instantiiert werden, sie dienen also nur der Erweiterung von nodes. Beispielsweise wird das Mixin *mix:referenceable,* einem Node Type hinzugefügt, wenn dieses referenzierbar sein soll. Dieses Mixin hat folgende in der CND Notation aufgeführte Definition:

```
[mix:referenceable] mixin
- JCR:uuid (STRING) autocreated mandatory protected initialize
```

Dieses Mixin erweitert den jeweiligen node type um eine automatisch generierte UUID[10]. Referenzierende nodes benötigen nur die UUID eines nodes um die Referenz aufzubauen. In dem Abschnitt 2.2.4.2 wird die hier verwendete Notation erläutert.

Vererbung Alle Primary Node Types erben von mindestens einem anderen Primary Node Type. Als Basis dient der Node Type *nt:base* alle anderen nodes erben von dieser. Mehrfachvererbung ist erlaubt und üblich. Hier gibt es nur eine Einschränkung, die die Vererbung

[10]Jackrabbit verwendet die UUID in der Version 4. Hierbei handelt es sich um eine 16-Byte Zahl mit einem Wertebereich der Kardinalität von 2^{122}. Eine UUID wird zufällig generiert, die Eindeutigkeit wird dadurch erreicht, dass die Wahrscheinlichkeit von zwei identischen Zufallszahlen in diesem Wertebereich gegen Null tendiert.

von Attributen wie z.B. orderable betrifft. Diese werden zur Vermeidung von Mehrdeutigkeiten nicht vererbt. Ansonsten erben Subtypen alle properties, child nodes und Mixins ihrer Supertypen. Mixins können, müssen aber keinen Supertype haben.

Child Nodes Node Types können keine oder mehrere child nodes enthalten. Dieses Konzept entspricht einer Has-A Beziehung aus der objektorientierten Programmierung. Alle properties und child nodes eines child nodes stehen im Node Type zur Verfügung.

2.2.4.2 Node Type Notation

In der Modellierungssprache „Compact Namespace and Node Type Definition“ (CND) werden primary node types, properties und Mixins, also die zentralen Datenstrukturen definiert. [11] Hierbei wird eine Definition in einer CND Datei abgelegt. Eine Definition beinhaltet eine Reihe von Namensraum Deklarationen, die node type Definition, child nodes und Mixins. Folgende Erläuterung behandelt nur eine Teilmenge der CND, deckt aber alle in Abschnitt 6.1.1 verwendeten Teile der Sprache ab.

```
1  <ns = 'http://namespace.com/ns1'>
2  <ex = 'http://namespace.com/ns2'>
3
4  //Kommentar
5
6  /* Ein
7   * Kommentar über
8   * mehrere Zeilen
9  */
10
11 [ns:example] > mix:referenceable, ex:supertype orderable mixin
12 - ex:property (string) = 'defaultValue' autocreated mandatory
       protected initialize
13
14 - ex:property2 (int)
15
16 + ns:node (ns:reqType1, ns:reqType2)
```

Abbildung 2.5: Definition eines Node Types

[11] Im Zuge des JSR 283, wird ein Ansatz entwickelt, der die Typ Definitionen in XML ausdrückt.

Zeile 1-2: Es werden die Namensräume *ns* und *ex* definiert und jeweils einer URI zugeordnet.

Zeile 4, 6-9: Erläutern die Deklaration von Kommentaren.

Zeile 11: Der Bezeichner des node_types, der hier definiert wird, lautet *example* und gehört dem Namensraum *ns* an. Er hat das Mixin *mix:referenceable* und erbt von der node *ex:supertype*. Durch die Attribute *orderable* und *mixin* wird ausgedrückt, dass es sich um ein Mixin handelt und dass seine child nodes eine festgelegte Reihenfolge besitzen.

Zeile 12: Hier wird ein property definiert. Es hat den Bezeichner *property* ist vom Typ String und hat den Standardwert *defaultValue*. Ihre Attribute haben folgende Bedeutung:

mandatory: Es handelt sich um ein Pflichtfeld.

autocreated: Das property wird automatisch generiert wenn die Eltern node erzeugt wird.

protected: Das property kann weder entfernt noch verändert werden.

initialize Das property wird erneut gesetzt wenn eine neue Version der node erzeugt wird.

Zeile 14: Eine property mit dem Bezeichner *property2*, dem Typ *int* und dem Namensraum *ex* wird definiert.

Zeile 16: Hierbei handelt es sich um eine child node mit dem Bezeichner *node*. Diese muss mindestens von den beiden Typen *ns:reqType1* und *ns:reqType2* sein.

2.3 OSGi

Die OSGi Service Plattform ist eine von der OSGi Alliance spezifizierte Java-basierte Komponentenplattform. Es erlaubt die Entwicklung von Java Applikationen in Form von Modulen, welche in dieser Terminologie Bundles genannt werden. Darüber hinaus bietet es Unterstützung für ein serviceorientiertes Programmiermodell.
Das Framework verwaltet die Installation, die Updates und den Lebenszyklus der Bundles. Dabei wird eine hohe Flexibilität und Skalierbarkeit erreicht. Bundles können zur Laufzeit durch Fernwartung hinzugefügt, gestartet, konfiguriert und deinstalliert werden. Auch werden die Abhängigkeiten (dependencies) der Bundels von OSGi verwaltet. Insbesondere können so in Konflikt stehende Abhängigkeiten von Bundels gehandhabt und die Konflikte aufgelöst werden. Innerhalb einer OSGi Service Plattform laufende Applikationen können durch Fernwartung administriert werden. Bei Apache Felix steht hierfür die „Apache Felix Web Management Console“ zur Verfügung.
Ein Bundle bietet seine Funktionalität in Form eines oder mehrerer dynamischer Services an. Diese Services werden an einer zentralen Stelle, der Service Registry, registriert und können so von Service-Verbrauchern in anderen Bundles gefunden und verwendet werden.

Historie Die Spezifikation von OSGi wird von der OSGi Alliance erstellt. Diese ist ein Zusammenschluss von ursprünglich knapp 30 namhaften IT-Unternehmen und besteht seit dem März 1999. Sie geht zurück auf den JSR-8: „Open Services Gateway Specification” und wurde gegründet um den Anforderungen im Bereich der embeded devices gerecht zu werden (vgl. (Kri08), S. 3). Das Hauptaugenmerk der Spezifikation lag bei den „Service-Gateways“, Servern die lokale Netzwerke mit externen verbinden und so als Brücke und Service Provider fungieren. Die Abkürzung OSGi stand ursprünglich für „Open Services Gateway initiativ“, dieser Begriff ist allerdings veraltet und findet heute keine Verwendung mehr.
Die Anforderungen in den embeded devices bestanden darin, dass unabhängige Teams von Entwicklern ihre Produkte auf der gleichen Hardware ausführen mussten. Die Spezifikation liegt zur Zeit in der Version 4.2 vor. Zunächst wurde das Framework in der Telematik, der Gebäudeautomatisierung und der Automobil Branche verwendet. 2003 wurde die Spezifikation um eine mobile Java Service Plattform (JSR 232) erweitert. 2004 wurde die Spezifikation von mehreren open source communities aufgenommen, wie Eclipse Equinox, Apache Felix und Knopflerfish (vgl. (OSG09a)). Z.B. arbeitet die IDE Eclipse [12] seit der Version 3.0 mit Equinox, einer OSGi Implementierung. So werden Eclipse Plugins als OSGi Bundles entwickelt und eingesetzt.

[12] `http://www.eclipse.org/`

Modularität Eines der grundlegendsten Konzepte von OSGi ist das der Modularisierung. Dieses wird im Folgenden vorgestellt.
Ein Modul ist eine Komponente eines größeren Systems. Ein Modul zeichnet sich durch zwei Haupteigenschaften aus; seinen starken Zusammenhalt sowie seine geringe Kopplung. Unter starkem Zusammenhalt versteht man die Eigenschaft des Moduls, für exakt eine Aufgabe zuständig zu sein und diese zu erfüllen. Es enthält keine Funktionalität, die sich nicht auf diese eine Aufgabe bezieht. Somit haben Module mit einem starken Zusammenhalt eine feine Granularität, sind robust, wiederverwendbar und einfacher zu verstehen. Der Zusammenhalt ist also eine interne Metrik. Im Gegensatz dazu handelt es sich bei der Kopplung um eine externe Metrik. Diese gibt an wie stark das Modul an andere Module gebunden ist. Lose gekoppelte Module haben nur eine exakte Schnittstellen-Vereinbarung. Darüber hinaus gehende Vereinbarungen, sind nicht zulässig. Eine solche Vereinbarung könnte z.B. darin bestehen, dass eine Collection als Rückgabewert einer Methode nach einem bestimmten Kriterium sortiert ist. Änderungen der Implementation einer Schnittstelle sollen keine Änderungen eines anderen Moduls nach sich ziehen. Durch diese Eigenschaften ist ein modulares System leicht änderbar. Ein Modul kann gegen ein anderes ausgetauscht werden.
Auch kann ein System leichter verstanden werden, indem induktiv die Module erarbeitet werden und sich daraus das Gesamtbild ergibt. Um parallele Entwicklung zu ermöglichen können unterschiedliche Teams mit der Entwicklung von Modulen beauftragt werden. Die konkrete Funktionsfähigkeit eines Moduls kann durch Testfälle überprüft werden. Außerdem ist die Fehlersuche und Behebung effizienter, wenn der Fehler in einem abgeschlossenen Modul auftritt. Durch die Wiederverwendung von Modulen kann eine hohe Flexibilität erreicht werden.

Die Modularisierung ist ein altbewährtes Konzept zur Bewältigung von Komplexität. So fassen Richard Gauthier und Stephan Ponto 1970 in ihrem Buch „Designing Systems Programs“ zusammen:

> „A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined; there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can

be traced to specific system modules, thus limiting the scope of detailed error searching.“(GP70)

Mit dem Aufkommen des objektorientierten Programmierparadigma zu Beginn der 80er Jahre hatte man bereits mit einer hohen Wiederverwendbarkeit auf Klassenebene gerechnet. Auf Grund der hohen Kopplung von Klassen untereinander ist eine Wiederverwendung nur sehr begrenzt praktikabel (vgl. (Kri08), S. 45; (OSG09c), S. 633). Eine Anstrengung diese Kopplung zu reduzieren, ist z.B. in Java das Konzept der Interfaces. Ein Interface beschreibt dabei nur die Schnittstelle über welche die Klassen miteinander kommunizieren. Implementierungsdetails sind nicht bekannt. Zur Entwicklungszeit reicht es aus eine Abhängigkeit durch ein Interface zu beschreiben. Erst zur Laufzeit wird eine konkrete Implementierung gebunden. Dabei besteht nur ein weiteres Problem, nämlich, dass eine Klasse ihre Abhängigkeiten kennen muss um diese zur Laufzeit zu erzeugen. Dieses Problem wird durch das Konzept von Dependency Injection gelöst. Die Abhängigkeiten einer Klasse werden an einer zentralen Stelle konfiguriert. Hierdurch wird eine geringe Kopplung ermöglicht. Dennoch fehlen dynamische Ansätze in einem solchen System.

2.3.1 Schichten von OSGi

OSGi core besteht aus folgenden aufeinander aufbauenden Schichten (vgl. (OSG09b), S. 1; (OSG07), S. 5, S. 11) :

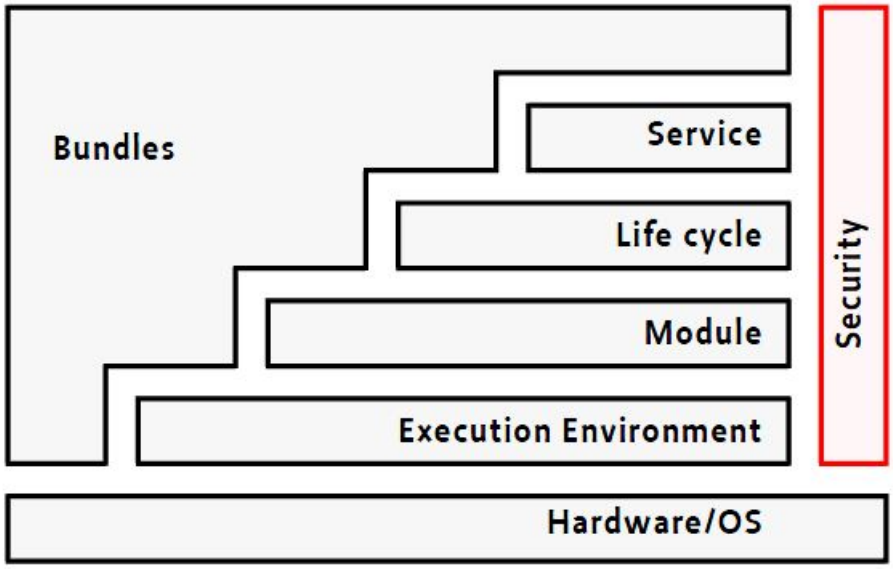

Abbildung 2.6: OSGi Schichten((OSG09b), S. 3)

Security Layer: Die Sicherungsschicht bildet einen Rahmen und setzt auf die Java 2 Security Architektur auf. Durch policy Dateien können Zugriffsrechte auf Bundle Ebene

definiert werden. Außerdem können Bundles digital signiert werden um die Herkunft dieser zu verifizieren und ein Installieren eines nicht signierten Bundles zu verhindern.

Execution Environment: Die OSGi Spezifikation basiert auf der Java Virtual Machine. Die JVM bietet eine bewährte sichere, offene und portable Umgebung. Abhängig von den gegebenen Voraussetzungen variiert die verwendete JVM. Z.B. wird Java sehr erfolgreich in mobilen Geräten wie Smartphones oder Handys eingesetzt. Bereits 2007 existierten weltweit über 1 Milliarde Handys auf denen Java Applikationen ausgeführt werden konnten. In diesen Umgebunden wird als Execution Environment die Java Micro Edition (Java ME) verwendet.

Module Layer: Bietet das Konzept zur Modularisierung von Java Applikationen durch Bundles an. Siehe hierzu den Abschnitt 2.3.2.

Life Cycle Layer: Stellt eine API zum Administrieren des Lebenszyklus von Bundles bereit. Diese wird sowohl von Bundles verwendet als auch von Managementkonsolen zur Remoteadministration. Der Lebenszyklus eines Bundles wird in 2.8 behandelt.

Service Layer: Diese Schicht enthält die Service Registry. Hier werden die Services zentral registriert und abgerufen. Ein Bundle kann eine beliebige Anzahl von Services registrieren. Services werden in dem Abschnitt 2.3.3 behandelt.

Bundles: In dieser Schicht sind die anwendungsspezifischen Bundles angesiedelt.

In Abbildung 2.6 wird die Hierarchie der Schichten deutlich. Die Bundles und der Security Layer kommunizieren mit allen übrigen Schichten.

2.3.2 OSGi Bundle

Ein OSGi Bundle ist eine JAR[13] Datei, deren `MANIFEST.MF` Datei spezielle deskriptive Metadaten enthält. Diese befindet sich innerhalb des Ordners `META-INF`. In Abbildung 2.7 ist eine Beispiel Manifest Datei aufgeführt (vgl. (Rub09), S. 6; (Bar09a), S. 25; (OSG09b), S. 39 ff).

[13] Sun Microsystems, Java Archive (JAR) files, `http://java.sun.com/j2se/1.5.0/docs/guide/jar/`

```
1 Manifest-Version: 1.0
2 Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
3 Bundle-Name: Hello World
4 Bundle-SymbolicName: helloworld
5 Bundle-Description: A Hello World bundle
6 Bundle-Version: 1.0.0
7 Bundle-Activator:org.example.Activator
8 Export-Package:org.example;version="1.0.0"
9 Import-Package: org.example.dependencies;version="1.3.0"
```

Abbildung 2.7: `MANIFEST.MF`

Bundle-Name: Ist der Name des Bundles.

Bundle–Symbolic Name: Dieses Attribut ist ein Pflichtfeld und muss angegeben werden. Es bildet zusammen mit dem Attribut Bundle-Version eine eindeutige Id.

Bundle-Description: Ist eine menschenlesbare Beschreibung über die Funktionalität und den Kontext des Bundles.

Bundle-Version: Ordnet dem Bundle eine Versionsnummer zu.

Bundle-Activator: Spezifiziert die Java Klasse, welche gestartet wird, wenn das Bundle aktiviert wird.

Export-Package: Die angegebenen Java Packages werden exportiert und so weiteren Bundles zur Verfügung gestellt.

Import-Package: Spezifiziert die von dem Bundle benötigten Packages. Dabei kann eine Version oder ein Bereich von akzeptierten Versionen definiert werden.

Zu jedem Bundle gehört ein spezieller Class Loader. Dieser ist für das Erstellen und die Validierung der Bundles sowie deren Installation zuständig. Weiterhin enthält ein Bundle eine Reihe von Class Dateien und Ressourcen, wie Konfigurationsdateien und Bibliotheken in Form von JAR Dateien. Die Zeilen eins und zwei in Abbildung 2.7 sind standardmäßig in jeder JAR Datei enthalten. Dadurch, dass ein OSGi Bundle nur zusätzliche Informationen in der `MANIFEST.MF` Datei hält, kann dieses auch außerhalb der OSGi Service Plattform verwendet werden.

2.3.2.1 Lebenszyklus von Bundles

Jedes Bundle folgt einem definierten Lebenszyklus. Das Aktivitätsdiagramm in Abbildung 2.8 veranschaulicht diesen.

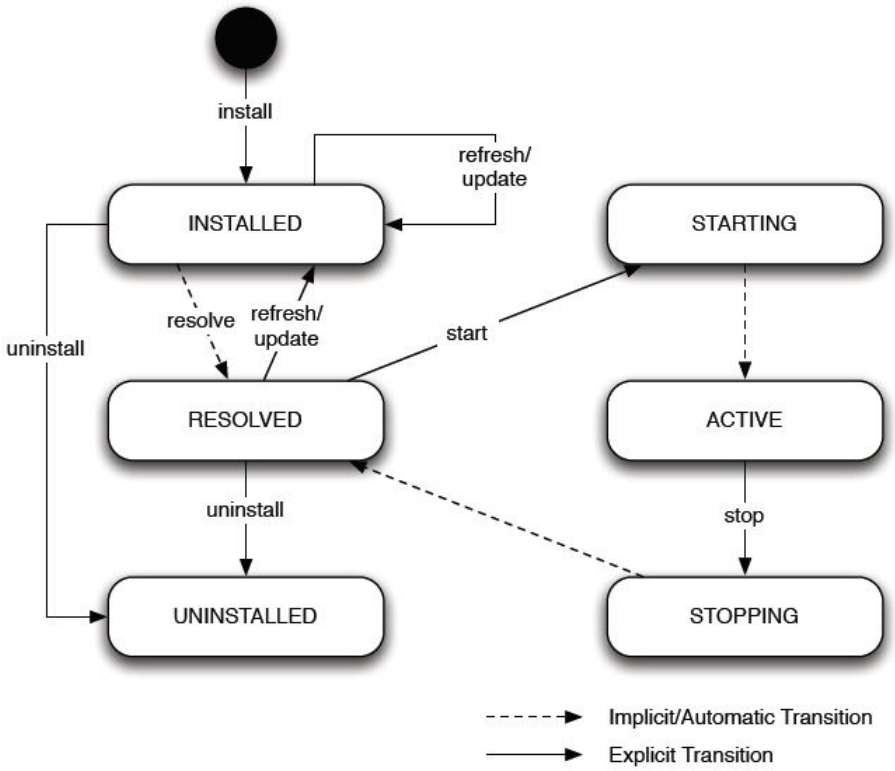

Abbildung 2.8: Lebenszyklus eines Bundles ((Bar09a), S. 37).

Ein Bundle beginnt seinen Lebenszyklus damit, dass es installiert wird. Dies geschieht bei der Implementation Apache Felix z.B. in der Eingabeaufforderung durch den Befehl *install file:<filename>.jar* oder auch in der „Apache Felix Web Management Console“ (WMC). Nach der Installation wird überprüft ob das Bundle ausgeführt werden kann. Es muss eine entsprechende Java Umgebung vorhanden sein, die zu importierenden packages und die benötigten Bundles müssen verfügbar sein und in dem Zustand Resolved vorliegen.
Sind diese Anforderungen erfüllt, findet ein impliziter Statusübergang in den Zustand Resolved statt. Wird ein Bundle durch den Befehl *start <idDesBundels>* oder durch die WMC gestartet, werden zunächst die Abhängigkeiten durch den class loader geladen. Anschließend wird das Bundle gestartet und in den Zustand Starting überführt. Dieser Status drückt aus, dass das Bundle im Begriff ist zu starten. Nach dem erfolgreichen Laden der Abhängigkeiten wird das Bundle implizit in den Zustand Active überführt. Wird der *stop* Befehl ausgeführt geht das Bundle in den Zustand Stopping. Hier können Aufräumarbeiten stattfinden z.B. kön-

nen bestehende Sperren gelöst werden. Anschließend findet wieder ein impliziter Übergang in den Zustand Resolved statt. Durch den *uninstall* Befehl geht das Bundle in den Zustand Uninstalled über, kann aber durch erneutes Installieren den Lebenszyklus erneut durchlaufen. Dieser Zustand symbolisiert, dass ein Bundle aus dem Lebenszyklus ausgetreten ist. Es ist für die OSGi Service Plattform nicht mehr existent. Bei erneutem Installieren wird eine neue Id vergeben.

2.3.3 Services in OSGi

In OSGi ist ein Service ein POJO (Plain Old Java Object), welches in der Service Registry unter dem voll qualifizierten Namen eines oder mehrerer Interfaces registriert wurde, die es implementiert. Dieser Service wird von anderen Services verwendet und wird von diesen mit dem Namen eines Interfaces bei der Service Registry erfragt. Die Services in OSGi sind eine Variante von Service Oriented Computing (SOC), auch vergleichbar mit den Services im WAM (Werkzeug-Material-Automaten) Architekturstil oder jenen in Service Oriented Architecture (SOA). Im Gegensatz zu SOA handelt es sich hierbei aber nicht um ein verteiltes, sondern um ein lokales Konzept, dessen Reichweite sich auf eine JVM beschränkt (vgl. (Bar09a), S. 78). Generell bieten serviceorientierte Architekturen eine Reihe von Vorteilen wie (vgl. (DJC08), S. 1; (EHL07), S. 1-2; (Erl09), S. 49):

Lose Kopplung: Es wird immer gegen eine Schnittstelle programmiert. Die konkrete Implementierung kann zur Laufzeit ausgetauscht werden.

Hohe Kohäsion: Ein Service sollte so geschnitten sein, dass er genau eine Verantwortlichkeit hat.

Abstraktion: Nach dem Entwurfsprinzip „Divide and Conquer“ wird das Gesamtproblem in Teilziele unterteilt und diese bearbeitet. Bei der Komposition der Teile wird von den Implementierungsdetails auf die Benutzung der Komponente abstrahiert.

Späte Bindung: Erst zur Laufzeit werden Services bei einem Service Broker erfragt und gebunden.

Geheimnisprinzip: Die Details der Implementierung eines Services sind dem Service Verbraucher unbekannt.

Dynamik (Komposition): Neue Funktionalitäten können durch die Wiederverwendung bestehender Services schnell etabliert werden.

Zustandslosigkeit: Einzelne Services arbeitet zustandslos und arbeiten immer entsprechend des Service-Vertrages.

Reduzierung von Risiken: Durch die Wiederverwendung von getestetem Code und bestehender Laufzeit Umgebung.

Kostenreduzierung: Verhinderung von Code Dubletten durch Wiederverwendbarkeit.

Iterative Zyklen: Eine iterative Weiterentwicklung der Gesamtarchitektur wird durch eine iterative Erweiterung der Komponenten oder einzelnen Services entsprechend des Service-Vertrags erreicht.

2.3.3.1 Statisches Dependency Injection

Dependency Injection (DI) ist ein Entwurfsmuster. Es erlaubt die Abhängigkeiten von Klassen auszulagern und zentral an einer Stelle zu konfigurieren. In den meisten objektorientierten Programmiersprachen existieren entsprechende Frameworks, die dieses bewerkstelligen. In Java ist ein weit verbreitetes der IoC (Inversion of Control) container des Spring Frameworks[14].
Generell unterscheidet man nach Martin Fowler hierbei zwischen drei Typen (vgl. (Fow04)):

Typ 1: Constructor Injection Die Abhängigkeiten einer Klasse werden bei der Instantiierung eines Objektes als Parameter an den Konstruktor übergeben.

Typ 2: Setter Injection Die Abhängigkeiten einer Klasse werden nach der Instantiierung des Objektes durch setter - Methoden dem Objekt übergeben. Erst nachdem alle Abhängigkeiten gesetzt sind, kann das Objekt verwendet werden.

Typ 3: Interface-Injection Durch die Implementierung eines Interfaces werden Methoden gekennzeichnet, welche verwendet werden, um die Abhängigkeiten von außen einem Objekt zuzuführen.

Das Spring Framework verwendet wahlweise die Typen 1 und 2. Der IoC container arbeitet nach dem Hollywood - Prinzip: „Don't call us we call you". Die Java Objekte werden in einer XML-Konfigurationsdatei mitsamt ihrer Abhängigkeiten als Beans deklariert. Diese werden in ihrer Verwendung nicht direkt erzeugt, sondern durch den IoC container. Der IoC container liegt in Form der Klasse *BeanFactory* oder *ApplicationContext* vor. Die Abhängigkeiten werden bei allen DI Frameworks und allen drei Typen statisch deklariert, sind so fest zugeordnet und zur Laufzeit nur noch manuell änderbar. Dies bringt einige Probleme mit sich, z.B. muss der Abhängigkeitsgraph gehandhabt werden, welcher zur Startzeit dadurch entsteht, dass z.B. Objekt B Objekt A benötigt und aus diesem Grund Objekt A vor Objekt B erzeugt werden muss. Bei mehreren tausend Objekten kann dies sehr komplex werden. Außerdem kann ein Update eines Moduls, einer Abhängigkeit, nicht ohne einen Neustart der Applikation erfolgen.

[14] `http://static.springsource.org/spring/docs/2.5.x/reference/beans.html`

2.3.3.2 Dynamische Services

Da sich Services an der Service Registry registrieren und abmelden können, handelt es sich hierbei nicht um eine statische Verknüpfung, wie sie durch DI erzeugt wird, sondern um eine dynamische zur Laufzeit veränderbare. Auch bei der Erzeugung der POJOs, welche die Services anbieten, muss der Abhängigkeitsgraph nicht beachtet werden. Um das Beispiel aus Abschnitt 2.3.3.1 aufzugreifen, kann auch Objekt B vor Objekt A erzeugt werden. Wenn der Service den Objekt A anbietet durch seine Registrierung bei der Service Registry verfügbar ist, wird Objekt B darüber benachrichtigt und kann diesen von nun an nutzen (vgl. (Bar09a), S. 92).[15]

> „Service-oriented architectures brought about new light to the research and practice of reusable components, because the “register, find, bind and execute” paradigm is a good style for software development. The “register, find, bind and execute” paradigm is common in our daily life, and it is successful in providing daily and on-line business services.“ ((Zhu05), S. 243)

Auch OSGi arbeitet nach dem „Registrierung, Finden, Binden und Ausführen“ Paradigma. Dieses wird in Abbildung 2.9 veranschaulicht.

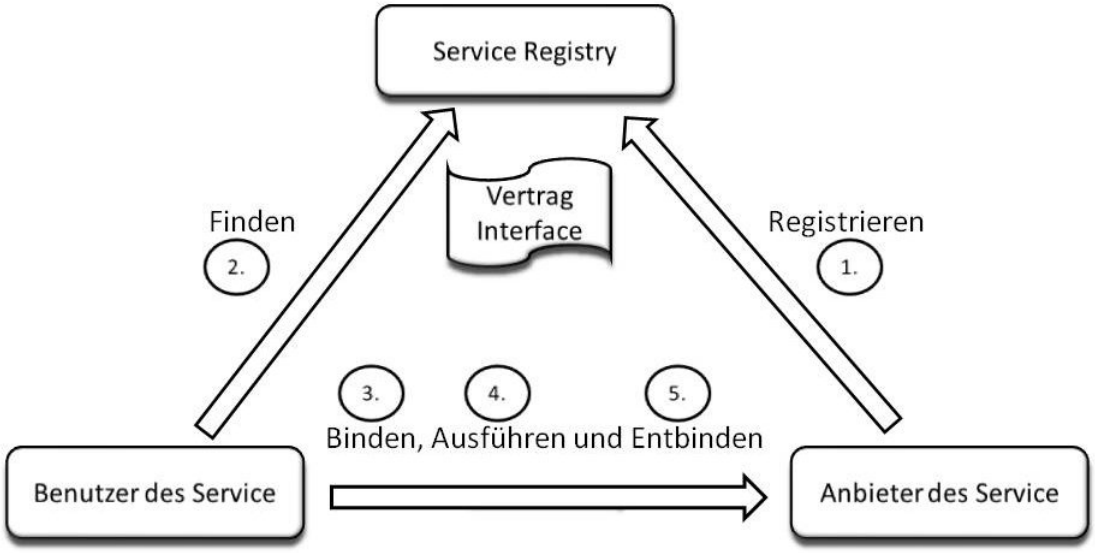

Abbildung 2.9: Das „Registrierung, Finden, Binden und Ausführen“ Paradigma (angelehnt an (Mah05)).

[15] Der Mechanismus, der hierfür verantwortlich ist heißt, Tracking Service und wird u.a. durch die Klasse *ServiceTracker* angeboten.

2.3.4 Der Java Class Loader in OSGi

Der Begriff Klassenpfad (classpath) kommt ursprünglich von einer Kommandozeile beim Aufruf eines Java-Programms. Er gibt einen oder mehrere Pfade zu JAR Dateien und Verzeichnissen an, in denen sich Class und Ressourcen Dateien befinden[16]. Diese Ressourcen nutzt die Klasse *java.lang.classloader* um Referenzen auf Klassen aufzulösen. Bei einem Fehlversuch wird eine *java.lang.ClassNotFoundException* geworfen und das Programm beendet. Ein Aufruf einer Java Applikation mit der Angabe eines Klassenpfades erfolgt unter UNIX wie folgt :

```
java -classpath log4j.jar:classes org.example.HelloWorld
```

Die Klasse *java.lang.ClassLoader* ist für das Laden von Klassen verantwortlich und hat zwei Hauptaufgaben:

1. Das Finden der Klassen in dem Klassenpfad anhand des voll qualifizierten Namens der Klasse.
2. Das Überführen der Class-Datei in ein Klassen Objekt im Hauptspeicher.

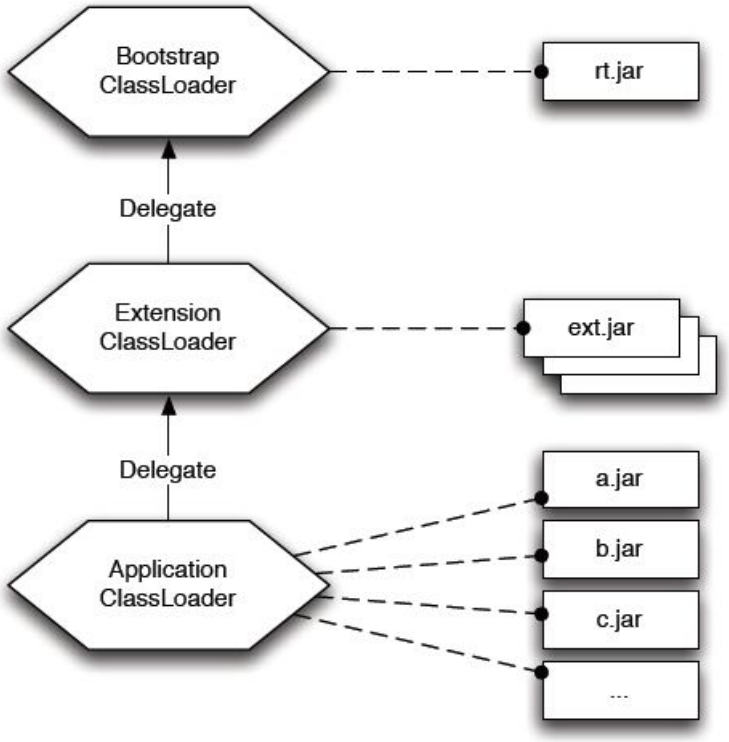

Abbildung 2.10: Die Standard Class Loader Hierarchie (vgl. (Bar09a), S. 7).

[16] Tatsächlich wird die Systemvariable CLASSPATH überschrieben.

Der erste Punkt kann durch das Ableiten der Klasse *java.lang.ClassLoader* überschrieben werden, um beispielsweise Code aus einem Netzwerk nachzuladen. Der zweite Punkt wird aber von der Methode *defineClass* bereitgestellt. Deren beiden Modifier *native* und *final* sind. So kann deren Verhalten nicht verändert werden. Obwohl es durch individuelle Class Loader möglich ist, unterschiedliche Versionen einer Klasse im Klassenpfad zu handhaben, ist dies nicht praktikabel. Dadurch würden die Class Loader zu einem elementaren Bestandteil der Applikation, welcher sehr schwer wartbar ist (vgl. (Rub09), S. 4).
Weiterhin existiert eine nicht veränderliche Hierarchie von ClassLoader. Bevor von einem ClassLoader nach einer Klasse gesucht wird, wird der Aufruf an die übergeordnete Hierarchie Stufe delegiert. So wird eine Klasse immer von dem höchst möglichen Punkt in dieser Hierarchie gefunden. Abbildung 2.10 veranschaulicht dies (vgl.(Bar09a), S. 7; (SM)).

Bootstrap ClassLoader: Lädt alle JRE Klasse, wie z.B. `rt.jar`.

Extension ClassLoader: Lädt alle Klassen, welche nicht direkt zu dem JRE gehören sich aber in dem Verzeichnis `jre/lib/ext` der JRE Installation befinden.

Application ClassLoader: Lädt die Klassen aus dem Klassenpfad.

2.3.4.1 JAR Hell

Durch diese Mechanismen hat eine Java Applikationen einen globalen Klassenpfad, der für alle Komponenten Gültigkeit hat. Daraus ergeben sich eine Reihe von Problemen. Zusammengefasst sind diese als die „Jar Hell“ bekannt (vgl. (Rub09), S. 3 ff.; (Bar09a), S. 5 ff.).

Der globale Klassenpfad: Der Klassenpfad wird von den Class Loadern als flacher Suchraum wahrgenommen. Existieren mehrere unterschiedliche Versionen einer Klasse im Klassenpfad, ist deren Auswahl zur Laufzeit nicht deterministisch. So kann es vorkommen, dass eine neuere Version von einer älteren verdeckt wird, was in der Praxis zu unverständlichem und nicht reproduzierbaren Verhalten führen kann. Dieses Problem ist besonders schwerwiegend, wenn unterschiedliche Komponenten, eine Abhängigkeit in unterschiedlichen Versionen benötigen.

Keine Modifikationen zur Laufzeit: Während die JVM läuft, ist es nicht möglich Class oder JAR Dateien auszutauschen, um neuere Versionen einzuspielen.

Keine Überwachung der Abhängigkeiten: Eine Abhängigkeit kann nur durch dessen voll qualifizierten Name repräsentiert werden. Bedingungen an eine Abhängigkeit wie, Komponente X ab Version 1.2 sind nicht abbildbar.

Verletzung des Geheimnisprinzips: Es existiert kein Mechanismus der das Geheimnisprinzip zwischen JAR Dateien sicherstellt.

Behandlung in OSGi
Jedes Bundle hat einen eigenen Class Loader mit einem separaten Klassenpfad. Dieser Mechanismus ist der Ansatzpunkt für das Importieren und Exportieren von packages, wie es im Abschnitt 2.3.2 auf Seite 21 beschrieben ist. Dadurch wird auch ein Konzept geschaffen, um das Geheimnisprinzip zwischen Komponenten zu gewährleisten. Um dieses Konzept umzusetzen, wird der Baum der Class Loader aus Abschnitt 2.3.4 in einen Graphen umgewandelt. Dies ist notwendig, damit die ursprünglichen Geschwister im Baum[17] einen unterschiedlichen Klassenpfad haben können. Weil in dem Baum immer zuerst die nächst höhere Hierarchiestufe befragt wird, bevor selbständig mit der Auflösung begonnen wird, ist es Geschwistern nicht möglich, einen unterschiedlichen Klassenpfad zu haben. In (Rub09) auf S. 7 ist auch im OSGi Kontext von einem Class Loader Baum die Rede. Dem widerspricht aber die Spezifikation (OSG09b) auf S. 37 und (Bar09a) auf S. 16ff.

2.3.5 Apache Felix

Apache Felix ist eine „OSGi R4 Service Platform“[18] Implementation und seit Anfang 2007 ein „top level project“ der Apache Software Foundation. Es ist quelloffen und steht unter der Apache License 2.0 frei zur Verfügung. Momentan steht es in der stabilen Version 2.0.0 zur Verfügung. Es handelt sich aber noch nicht um eine vollständige Implementation der Spezifikation. Mit dem Framework kann man über eine Eingabeaufforderung oder über die „Apache Felix Web Management Console“ (WMC) kommunizieren. Nach dem ersten Starten der Plattform sind die in Abbildung 2.11 ersichtlichen Bundles installiert.

```
Welcome to Felix ================
-> ps
START LEVEL 1
ID      State         Level   Name
[   0] [Active     ] [    0] System Bundle (2.0.0)
[   1] [Active     ] [    1] Apache Felix Bundle Repository (1.4.1)
[   2] [Active     ] [    1] Apache Felix Shell Service (1.4.0)
[   3] [Active     ] [    1] Apache Felix Shell TUI (1.4.0)
->
```

Abbildung 2.11: Bundles nach dem Starten von Felix. (Ausgabe des Befehls *ps*)

System_Bundle in diesem Bundle läuft das eigentliche Framework ab. Ein deinstallieren ist nicht möglich.

[17] Dies sind unterschiedliche Bundles, welche in der Hierarchie der Class Loader auf gleicher Ebene stehen.
[18] `http://www.osgi.org/Specifications/HomePage`

Apache Felix Shell Service stellt eine API zur Verfügung über die mit dem Framework kommuniziert werden kann. Sie kann von mehreren Schnittstellen verwendet werden, wie textuellen (Apache Felix Shell TUI) oder grafischen (Apache Felix Web Management Console).

Apache Felix Shell TUI stellt die textuelle Schnittstelle für das Bundle Apache Felix Shell Service dar. TUI steht für Textual User Interface.

Apache Felix Bundle Repository stellt ein Verzeichnis dar, welches Bundles über eine http Schnittstelle verfügbar macht. Die Adresse des Repositorys ist in der zentralen Konfigurationsdatei, *config.properites* in dem Ordner FELIX_HOME[19]\conf, konfigurierbar. Per dafault zeigt diese auf `http://felix.apache.org/obr/releases.xml`. Über dieses Verzeichnis lassen sich Bundles abfragen und installieren. Dieses Konzept vereinfacht das Installieren von Bundles und es ermöglicht auf einfache Weise existierende Bundles zu verwenden. z.B. kann durch den Befehl *obr start "Apache Felix Web Management Console"* das Bundle „Apache Felix Web Management Console“ aus dem Repository heruntergeladen und gestartet werden. Anschließend ist die WMC unter der URL `http://localhost:8080/system/console/bundles` erreichbar[20]. Zu beachten ist dabei, dass die Abhängigkeiten dieses Bundles automatisch aufgelöst, mit installiert und gestartet werden. Abbildung 2.12 stellt die installierten Bundels nach der Installation des Bundles WMC dar.

Weiterhin sehen wir in der Abbildung 2.11, dass jedes Bundle eine eindeutige id besitzt. Wie in 2.3.2.1 beschrieben, ist eine id nicht an ein Bundle gebunden, sondern wird für jede Installation eines Bundles neu vergeben. Alle Bundles befinden sich in dem Zustand Active. Das Level ist ein Start - Level und gibt an, in welcher Reihenfolge die Bundles gestartet werden. Das Level 0 ist für das System Bundle reserviert. Die Startreihenfolge von Bundles mit gleichem Start - Level ist nicht deterministisch. Dies ist auch ein Management Gesichtspunkt. Ein Bundle kann in einem Framework den Start Level 1 haben und in einem weiteren den Start Level 50.

[19] Hierdurch wird auf das Basis-Verzeichnis der Felix Installation verwiesen

[20] Die Initiale Benutzerkennung lautet admin/admin.

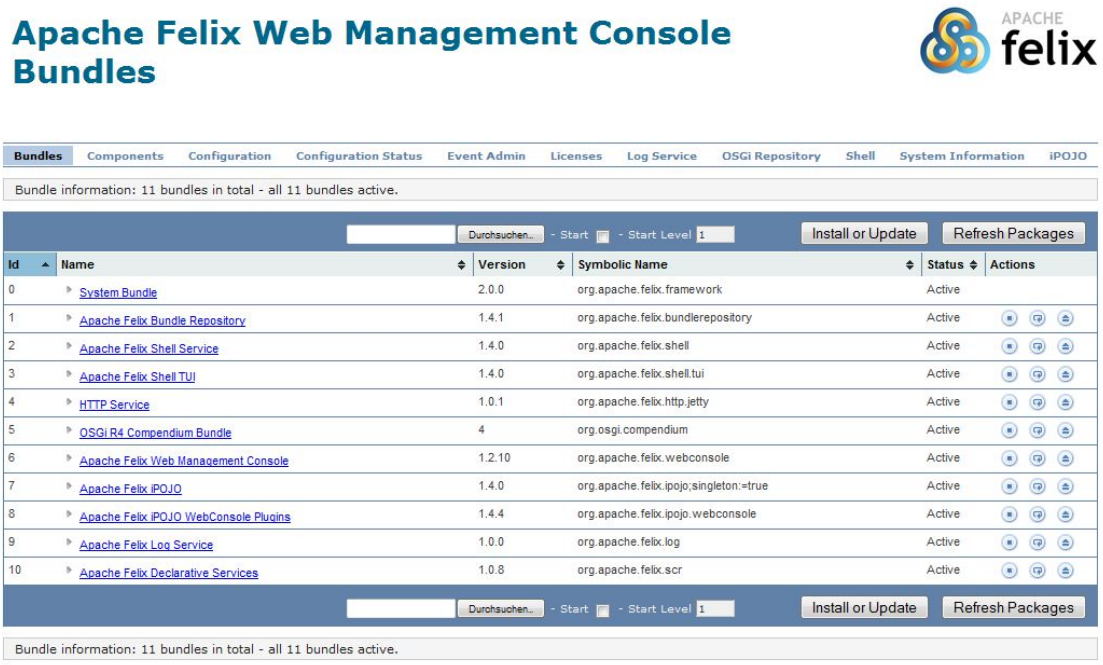

Abbildung 2.12: Apache Felix Web Management Console.

2.4 Apache Sling

Apache Sling ist ein quell offenes Web-Framework für JSR-170 kompatible Java Content Repositories. Es wurde zunächst als internes Projekt der Day Software Inc. entwickelt und als Teil ihres Produktes „Web Content Management“ eingesetzt. Am 27.08.2007 wurde die Code Basis der Apache Software Foundation zur Verfügung gestellt. Dort war es zunächst ein Unterprojekt von Apache Jackrabbit. Seit dem 18.06.2009 ist es ein „top level project“ von Apache.

Apache Sling wurde entwickelt um Content zentrierte Web-Anwendungen auf Basis eines JCR zu entwickeln. Unter anderem werden eine Reihe von Skript Sprachen wie JSP, ESP, Javascript, Ruby oder Velocity unterstützt, um Content aus dem JCR zu lesen oder zu manipulieren. Ferner können die anwendungsspezifischen Komponenten als OSGi Bundles installiert werden. Das gesamte Framework fungiert innerhalb einer eingebetteten OSGi Umgebung. Als OSGi Implementierung kommt Apache Felix zum Einsatz. Als JCR wird Apache Jackrabbit verwendet.
Ein wichtiger Aspekt von Sling ist die Verwendung des Konzepts Representational State Transfer (REST). Hierdurch werden JCR Nodes als HTTP Ressource zur Verfügung gestellt.

Um das Beispiel aus der Abbildung 2.4 aufzugreifen, wird so die node i, die im JCR unter dem Pfad `/a/d/i` verortet ist, beispielsweise unter der Adresse `http://localhost:8080/a/d/i` verfügbar gemacht. Wobei der Host Name natürlich variabel ist. Weiterhin ist es so möglich, durch einen HTTP POST Request an diese URL, Properties dieser Node zu schreiben. Beispielsweise wird durch folgenden Request die Property isOnline der Node `/news/kuche-content-nt:contentBase` auf den Wert true gesetzt.

```
POST /news/kuche-content-nt:contentBase HTTP/1.1
Host: localhost:8080
Content-Type: multipart/form-data
Content-Length: 27

kuche-content:isOnline=true
```

Weiterhin sind Aktionen für alle HTTP Requests wie z.B. GET, PUT und HEAD definiert. Dieses Konzept geht auf die Dissertation von Roy Thomas Fielding, „Architectural Styles and the Design of Network-based Software Architectures“ aus dem Jahr 2000 zurück (vg. (Fie00)).

Kapitel 3

Deklarativer Umgang mit dynamischen Services

Der hohe Grad an Dynamik ist für Entwickler schwer zu verwalten und bedeutet einen zusätzlichen Entwicklungsaufwand. Um diesen zusätzlichen Aufwand zu minimieren, haben unterschiedliche Institutionen eine Reihe von Lösungen entwickelt. In diesem Abschnitt wird auf den zusätzlichen Entwicklungsaufwand eingegangen, Lösungen vorgestellt und miteinander verglichen.

3.1 Programmatische Umsetzung von dynamischen Services

Die OSGi Spezifikation definiert Rahmenbedingungen für alle in Abbildung 2.9 dargestellten Aktionen. Um einen Service zu Beginn des Zustands *Active* zu registrieren ist es zunächst notwendig einen Bundle-Activator zu implementieren. Ein Bundle-Activator implementiert das Interface *org.osgi.BundleActivator* und muss in der `MANIFEST.MF` Datei des Bundles als Bundle-Activator angegeben werden. Das Interface schreibt die beiden im Folgenden dargestellten Methoden vor.

```
public void start(BundleContext context) throws Exception
public void stop(BundleContext context) throws Exception
```

Über den Parameter *contex*t vom Typ *BundleContext* kann auf die Service Registry zugegriffen werden. Alle zur Verfügung gestellten Services müssen ausgehend von dieser Methode an der Service Registry registriert werden. Weiterhin müssen alle von dem Bundle verwendeten Services entweder gebunden werden oder es muss ein ServiceTracker initialisiert werden, um bei Registrierung des entsprechenden Services an der Service Registry benachrichtigt zu werden. Für jeden Service ist ein ServiceTracker zu implementieren, der von der

Klasse *ServiceTracker* erbt und mindestens die Methode *addingService(ServiceReference reference)* überschreibt. Eine typische Implementation ist in Abbildung 3.1 dargestellt.

```
class MyServiceTracker extends ServiceTracker{

   public MyServiceTracker(BundleContext context) {
         super(context, IServiceProvider.class.getName(),null);
   }

   @Override
   public Object addingService(ServiceReference reference) {
      sp = (IServiceProvider) context.getService(reference);
      return sp;
   }

   @Override
   public void remove(ServiceReference reference) {
      System.out.println("ServiceProvider has gone.");
   }
}
```

Abbildung 3.1: Individueller ServiceTracker.

Durch die Parameter im Aufruf des *super* Konstruktors wird definiert, an welchem Service Interesse besteht. Ist er verfügbar, wird die Methode *addingService* aufgerufen. Über den *BundleContext context,* der in der Super - Klasse *ServiceTracker* implementiert ist und die *ServiceReference* ist es dann möglich den Service zu binden.
Bei der Registrierung von Services können noch Properties an den Service gebunden werden, welche diesen näher beschreiben. Beim Binden dieses Services kann dann über diese Properties gefiltert werden. Ferner können Services unter unterschiedlichen Service Strategien zur Verfügung gestellt werden. Services können optional sein und/oder eine Default Implementation besitzen. Durch diese und weitere Mechanismen können Services durch eine Vielzahl von Optionen registriert und gebunden werden. Diese werden alle programmatisch definiert und sind so Bestandteil der Codebasis. Es vermischen sich Mechanismen der serviceorientierten Architektur mit der Anwendungslogik. Dies widerspricht dem Entwurfsprinzip der Kapselung und erhöht den Wartungsaufwand einer solchen Anwendung. Weiterhin ergeben sich folgende Nachteile (vgl. (OSG09c), S. 297):

Start-Dauer: Alle Services werden beim Starten instantiiert und registriert. In Systemen mit vielen Bundles kann dies schnell zu einer inakzeptablen Start-Dauer führen.

Speicherverbrauch: Jeder Service wird beim Starten mit all seinen zugehörigen Klassen erzeugt und im Speicher gehalten. Selbst wenn der Service nie verwendet wird.

3.2 Declarative Services

Declarative Services ist seit der Version 4.0 ein Bestandteil der OSGi Spezifikation. Dieser Aspekt wird in den Teilen „Service Compendium“ und „OSGi Mobile Specification“ der Spezifikation behandelt. Durch die Verwendung von Declarative Services ist es möglich die SOC spezifischen Aspekte weitgehend in XML Konfigurationsdateien auszulagern. Diese Datei muss in der `MANIFEST.MF` unter dem Key *Service-Component* angegeben werden.

Als zentrales Konzept dienen Service Components. Diese bestehen aus einer Komponentenklasse und aus einer Komponentenbeschreibung. In der Komponentenklasse können Callback-Methoden definiert werden, welche z.B. beim Starten oder Stoppen des Bundles ausgeführt werden. Die Komponentenbeschreibung definiert alle Abhängigkeiten zu anderen Services und alle Services, die an der Service Registry registriert werden sollen. Der Lebenszyklus einer Component wird von der Service Component Runtime (SCR) verwaltet. Beim Starten einer Komponente überprüft die SCR, ob alle benötigten Services zur Verfügung stehen. Ist dies nicht der Fall wird die Instantiierung der Component so lange verzögert bis alle Abhängigkeiten erfüllt sind. Eine mögliche lange Startzeit wird durch die verzögerte Aktivierung von Services vermieden. In Abbildung 3.2wird die Komponentenbeschreibung dargestellt.

```
1  <?xml version="1.0"?>
2  <component name="component" immediate="true">
3     <implementation class="org.kuche.bachelor.MyComponent"/>
4     <service>
5        <provide interface="org.kuche.bachelor.api.IComponent"/>
6     </service>
7     <reference name="serviceProvider"
8        interface="org.kuche.bachelor.component2"
9        bind="bindMethod"
10       unbind="unbindMethod"
11       cardinality="0..1"
12    />
13 </component>
```

Abbildung 3.2: Komponentenbeschreibung.

3.3 Spring Dynamic Modules for OSGi(tm) Service Platforms

Spring Dynamic Modules ist ein Unterprojekt des Spring Frameworks. Dessen generelles Ziel ist es die Entwicklung von Java basierten Anwendungen zu vereinfachen und gute Programmierpraktiken zu fördern. Dazu bietet Spring eine Vielzahl von unterstützenden Entwicklungswerkzeugen. Auf die Umsetzung des Konzepts Dependency Injection wurde bereits in Abschnitt 2.3.3.1 eingegangen. Das Spring Framework ist sehr umfangreich und es existieren neben Spring DM noch eine Reihe von Unterprojekten, wie z.B. die Portierung des Frameworks auf die .net Basis oder Spring LDAP. Zu den Kernfunktionalitäten gehören neben dem DI eine Abstraktionsschicht für das Verwalten von Transaktionen, eine Abstraktionsschicht für JDBC, die Integration von Datenbank-Schnittstellen wie Toplink, JDO oder Hibernate und aspektorientiere Funktionalitäten. Auch ein Ansatz zur Entwicklung von Web Applikationen auf Basis des MVC Entwurfsmuster wird angeboten.
Spring Dynamic Modules wurde 2006 ins Leben gerufen und hat es sich zu Aufgabe gemacht die beiden Welten von Spring und OSGi. Das Spring DI Konzept und OSGi können als orthogonale Konzept angesehen werden (vgl. (Rub09), S. 109). Diese beiden Konzepte werden in Spring DM miteinander verbunden. So können Spring Beans aus dem DI Konzept als Services registriert werden. Pro Bundle wird dabei ein *ApplicationContext* verwendet. Beim Aktivieren des Bundles wird dieser gestartet und beim Deaktivieren zerstört. Der *ApplicationContext* stellt den IoC Container dar. Ein Bundle, das mit Spring DM erstellt wurde, unterscheidet sich von anders erzeugten Bundles nur dadurch, dass sie den Ordner `META-INF/spring` enthalten und dieser Spring Konfigurationsdateien im XML Format enthält. Die Deklaration der Services erfolgt innerhalb dieser Konfigurationsdateien. Dabei wird die Struktur der XML Dateien zur Beschreibung von DI Beans erweitert. Abbildung 3.3 zeigt ein Beispiel.

```
1  <bean id="guiBean"
2      class="org.kuche.bachelor.internal.Gui" init-method="start"
3      lazy-init="false" >
4  </bean>
5
6  <osgi:bundle id="starterId"
7      symbolic-name="symbolic" action="start">
8  </osgi:bundle>
9
10 <osgi:service id="GuiService" ref="guiBean"
11     interface="org.kuche.bachelor.IGui" depends-on="serviceProvider">
12     <property name="service">
13         <osgi:reference bean-name="serviceProvider"
14             interface="org.kuche.bachelor.MyService">
15             <osgi:listener ref="serviceProvider"/>
16         </osgi:reference>
17     </property>
18 </osgi:service>
```

Abbildung 3.3: Spring DM Konfiguration.

In den ersten vier Zeilen wird eine Bean definiert. Diese wird in Zeile zehn dazu verwendet einen Service zu registrieren. Der Service hat wiederum eine Abhängigkeit zu dem Service *org.kuche.bachelor.MyService*.
Einige Entwicklungswerkzeuge von Spring sind auch unter Spring DM nutzbar. Dies sind z.B. das MVC Entwurfsmuster für Web Applikationen oder die Aspektorientierte Programmierung. Weiterhin bietet Spring einen Enterprise Application Server auf Basis von OSGi an; den Spring dm Server. Dieser ist im Wesentlichen eine Portierung des Apache Tomcat Servers auf die OSGi Service Plattform mit einigen Erweiterungen im Bereich der Class Loader. Dieser bietet unter anderem die Möglichkeit Integrationstests für OSGi Anwendungen auszuführen und behebt Class Loading Probleme, die im Zusammenspiel von OSGi und Java Webapplikationen entstehen.
Die Verschmelzung der Spring DI Konzepte mit OSGi und die Möglichkeit weitere komplexe Entwicklungswerkzeuge zu verwenden bedeutet einen höheren und komplexeren Verwaltungs- und Konfigurationsaufwand. Dieser lohnt sich nur, wenn mindestens eine weitere Spring Technologie verwendet wird.

Spring DM als Bestandteil der OSGi Spezifikation

In der OSGi 4.2 Spezifikation wurde die Blueprint Container Specification aufgenommen. Die Spezifikation eines Dependency Injection Frameworks für OSGi (vgl. (OSG09c), S. 633 f.). Diese Spezifikation übernimmt Konzepte und Terminologien aus Spring DM. Einige Namen und Bezeichnungen wurden geändert, aber in wesentlichen Teilen ist Blueprint Service identisch mit Spring DM. Dies ist nicht überraschend, denn die Spezifikation wurde von Mitarbeitern des Unternehmens SpringSource entwickelt. SpringSource ist auch der Herausgeber des Spring Frameworks (vgl. (Wal09), S. 128). Durch diesen Umstand ist Spring DM auch gleichzeitig die Referenz Implementierung der Blueprint Spezifikation und momentan auch die einzige.

3.4 Google Guice peaberry

Guice ist ein Java Dependency Injection Framework von Google Inc. . In Guice werden Abhängigkeiten einer Klasse nicht über eine Auszeichnungssprache wie XML konfiguriert, sondern durch ein Konzept das mit Modulen und Bindings arbeitet. Die Konfiguration erfolgt über Annotations oder deklartiv. Ein Modul ist die Zusammenfassung von mehreren Bindings. Ein Binding ist dabei der Prozess eine Abhängigkeit, die durch ein Interface definiert ist, an eine konkrete Implementierung zu binden. Abbildung 3.4 zeigt ein Beispiel.

```
1 import com.google.inject.Binder;
2 import com.google.inject.Module;
3
4 public class AddModule implements Module {
5
6     public void configure(Binder binder) {
7          binder.bind(Add.class).to(SimpleAdd.class);
8         }
9 }
```

Abbildung 3.4: Bindings in Guice.

Wobei die Klasse *Add* das Interface darstellt und *SimpleAdd* eine Implementierung dessen ist. Module verwalten dabei eine Reihe von Bindings. In Zeile sieben wird ein Binding definiert. Die Module werden dann verwendet, um mit Hilfe von Injectors konkrete Instanzen zu erzeugen. Abbildung 3.5 zeigt ein Beispiel der Erzeugung einer Instanz der Klasse *SimpleAdd*.

```
import com.google.inject.Guice;
import com.google.inject.Injector;

public class AddClient {
        public static void main(String[] args) {
                Injector injector = Guice.createInjector(
                        new AddModule());
                Add add = injector.getInstance(Add.class);
        }
}
```

Abbildung 3.5: Instantiierung in Guice.

Peaberry ist eine erweiternde Bibliothek für Google-Guice. Momentan liegt es in der Version 1.1.1 vor. Es integriert das DI Konzept von Guice in OSGi. Dabei unterstützt es Dependency Injection für dynamische Services. Um peaberry zu verwenden muss ein entsprechendes Bundle installiert sein. In Abschnitt 3.6 wird veranschaulicht wie peaberry innerhalb von OSGi verwendet wird. Die hier verwendete Klasse AbstractModule implementiert das Interface Module, vereinfacht die Wiederverwendung von Modulen und es entstehen leichter lesbare Konfigurationen.

```
1  public void start(BundleContext bundleContext) {
2
3     Module module = new AbstractModule() {
4           protected void configure() {
5           install(Peaberry.osgiModule(bundleContext));
6           bind(WebApplication.class).toInstance(WebAppImpl.class));
7           bind(LoginContext.class).to(OWLoginChange.class);
8        }
9     };
10    injector = createInjector(module, getCustomModule());
11    Properties p = new Properties();
12    Object instance = injector.getInstance(WebApplication.class);
13    bundleContext.registerService(WebApplication.class.getName(),
14       instance, p);
15 }
```

Abbildung 3.6: peaberry in einer OSGi Umgebung.

Der obere Programmcode ist Teil des BundleActivators. Wie beim klassischen Umgang mit dynamischen Services (siehe Abschnitt 3.1) werden alle Services beim Starten des Bundles registriert.
Peaberry unterstützt momentan nicht die Konfiguration von Bundles und bietet keine Möglichkeit des Zugriffs auf den Lebenszyklus von Bundles.

3.5 iPOJO

iPOJO ist ein Unterprojekt von Felix bei der Apache Software Foundation. Es ist ein serviceorientiertes Komponenten Modell und hat das Ziel, die Entwicklung von dynamischen Services zu vereinfachen. Weiterhin verfügt iPOJO über einen erweiterbaren Mechanismus, der es erlaubt nicht funktionale Anforderungen zu handhaben und der die Interaktion von Services regelt. Ein Hauptziel ist es die Entwicklung von serviceorientierten Komponenten so nah wie möglich an der Entwicklung von POJOs zu halten. Der Code soll sich hierbei auf die Programmlogik beschränken und soll frei von SOC spezifischen Mechanismen und frei von nicht funktionalen Anforderungen sein (vgl. (EHL07)). iPOJO ist in einer großen Bandbreite von Anwendungsfällen anwendbar. So wird es unter anderem in Handys (J2ME Plattform), JEE Servern und der Gebäudeautomatisierung verwendet. Dadurch, dass es auf POJOs basiert, kann bestehender Code leicht verwendet und migriert werden. Um die Komponenten zu beschreiben, können entweder Annotation, XML oder eine API verwendet werden. Es kann mit beliebigen OSGi R4.1 Implementationen zusammenarbeiten. Dabei kommt IPOJO Core mit einer Größe von 200kb aus.

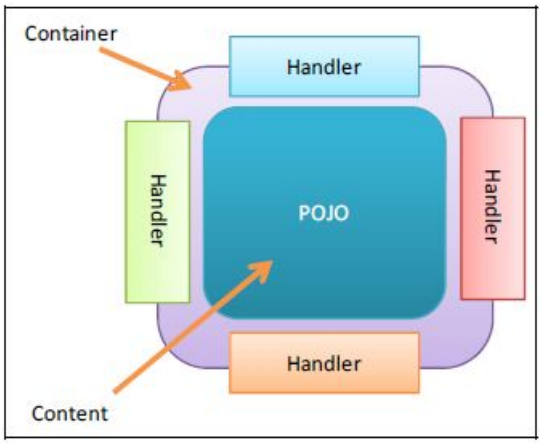

Abbildung 3.7: Ein iPOJO Container und seine Handler (vgl. (EHL07), S. 3; (iPOb)).

Der Begriff iPOJO steht für „injected POJO“ und spiegelt den generellen Ansatz von iPOJO wider, POJOs mit Hilfe von Handlern, welche nicht funktionale Anforderungen verwalten, in die OSGi Service Plattform zu injizieren. Tatsächlich wird der Bytecode von POJOs manipuliert, um ein installierbares OSGi Bundle zu erzeugen (vgl. (EHL07), S. 5). In Abbildung 3.8 ist dies der Schritt Manipulation. Bei dieser Aktion werden die Zeilennummern des bestehenden Codes beibehalten, damit die Log Ausgaben verlässlich bleiben. In diesem Zusammenhang spricht man von POJO Instanzen. Diese nehmen zur Laufzeit die beiden Status *invalid* und *valid* an. Deren Lebenszyklus ist in Abschnitt (3.5) beschrieben. Eine solche Instanz läuft in einem Container. Abbildung 3.7 stellt diesen Sachverhalt dar. Um diese Instanzen zur Laufzeit zu überwachen wurde eine Erweiterung für die WMC entwickelt. Das Bundle „Apache

Felix iPOJO WebConsole Plugins“. Mit dessen Hilfe können zur Laufzeit u.a. die Instanzen und deren Handler verwaltet werden.

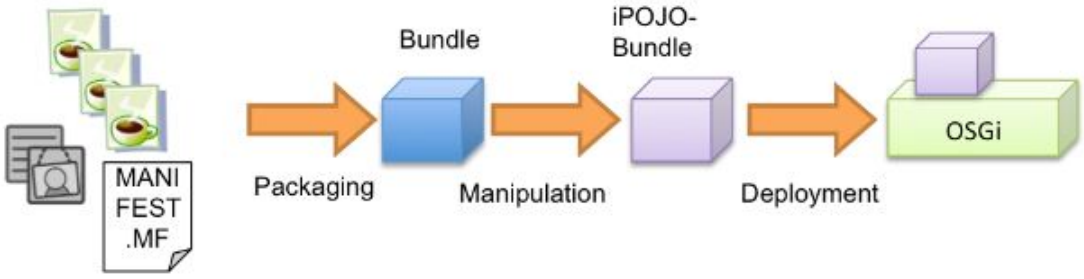

Abbildung 3.8: Vom Java Code zum OSGi Bundle (vgl. (iPOa)).

Diese Handler können selbst definiert werden; werden aber auch intern von iPOJO verwendet und zur Laufzeit dem POJO hinzugefügt. So kann z.B. die Veröffentlichung, das Finden und das Binden von dynamischen Services als nicht funktionale Anforderungen betrachtet werden. Tatsächlich existieren zwei Handler, welche die dynamische Interaktion der Services verwalten. Diese sind:

Provided Service Handler: Verwaltet die Veröffentlichung von Services.

Dependency Handler: Verwaltet das Suchen und Binden von Services.

In Abbildung 3.9 wird dieses Szenario veranschaulicht.

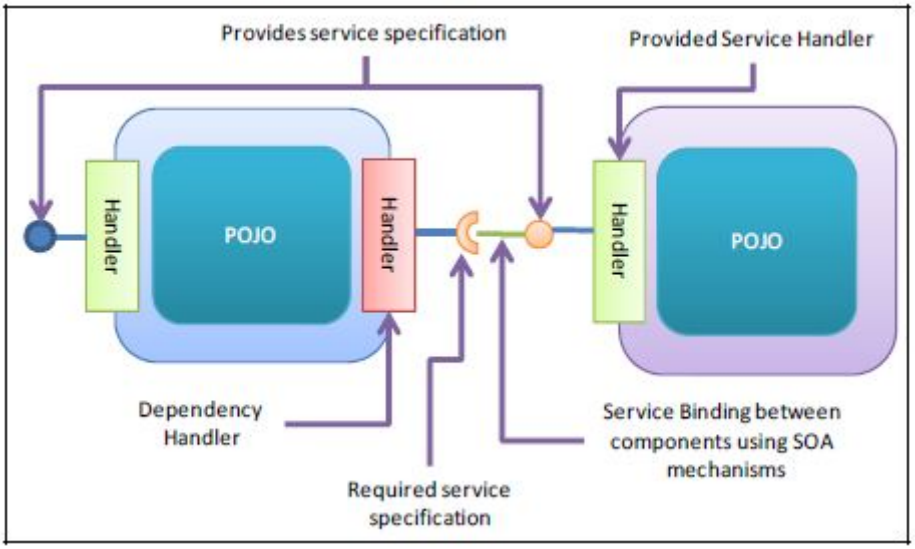

Abbildung 3.9: Verwaltung der dynamischen Services durch Handler. (vgl. (EHL07), S. 4).

Handler können dabei mit dem POJO oder untereinander kommunizieren.

OSGi und iPOJO

Die Spezifikation von OSGi überlässt die Verwaltung der Abhängigkeiten von Services den Entwicklern der Bundles. Diesen werden nur Werkzeuge wie die Klasse TrackingService an die Hand gelegt. iPOJO setzt hier auf der OSGi Service Plattform auf, ergänzt diese und nimmt den Entwicklern von Bundles die manuelle Verwaltung der Abhängigkeiten von Services ab.

Lebenszyklus von iPOJO Instanzen

Der Status einer iPOJO Instanz kann entweder *invalid* oder *valid* sein. Sie ist valide, wenn alle seine Handler valide sind. Beispielsweise kann ein Service einen weitern benötigen, der Dependency Handler kann diesen aber nicht finden, so bleibt der Dependency Handler invalide und somit auch die iPOJO Instanz. Sobald die iPOJO Instanz valide ist, wird sie gestartet. Es ist möglich Callback Handler zu definieren, welche Aktionen beim Starten und oder Stoppen der Instanz ausführen können.

Verteilte Services mit iPOJO

Obwohl es sich bei OSGi wie in Abschnitt 2.3.3 auf Seite 24 beschrieben um ein lokales Konzept handelt, ist es möglich mit Hilfe von iPOJO verteilte Services anzubieten und zu nutzen. Dabei wird die Technik der Web Services genutzt. Diese Möglichkeit von iPOJO wird im Weiteren nicht näher behandelt. Trotzdem ist es eine mächtige Erweiterung zu OSGi, welche nicht unerwähnt bleiben sollte.

3.6 Fazit

Das Ausgangsproblem war der Mehraufwand und die steigende Komplexität von OSGi basierten Anwendungen aufgrund der SOC spezifischen Mechanismen im Programmcode. Darüber hinaus wurde auf das Problem des Speicherverbrauchs und der Startdauer aufmerksam gemacht. Die hier vorgestellten Frameworks unterscheiden sich stark in der Breite ihrer Funktionalitäten. Peaberry und Spring DM verbinden beide die DI Konzepte mit dynamischen Services in OSGi. Darüber hinaus bietet Peaberry momentan keine weitere Unterstützung. Dagegen hat man mit Spring DM gleichzeitig Zugriff auf viele Spring Werkzeuge und kann auf einen speziellen Application Server für die Entwicklung von OSGi basierten Web Anwendungen zugreifen. Bezüglich des Speicherverbrauchs und der Startdauer bringt peaberry keine Verbesserung. Hingegen DS, iPOJO und Spring DM diese Situation verbessern. Mit iPOJO ist DI nicht möglich. Dafür bietet es eine Vielzahl von OSGi spezifischen Werkzeugen und Mechanismen.

Auch hinsichtlich der Beschreibung der Services und der SOC Mechanismen unterscheiden sich die Frameworks stark. Unter peaberry ist es weiterhin nötig Services programmatisch zu registrieren und zu binden. Unter DS und Spring DM kann diese Aufgabe durch eine entsprechende Konfiguration in XML erfolgen. In iPOJO erfolgt dies entweder in XML oder über Annotations.
Im Folgenden werden keine weiteren Spring Technologien verwendet, weshalb Spring DM aufgrund eines größeren Konfigurationsaufwand ausscheidet. Peaberry bietet nur die Verwendung von DI ohne weitere Unterstützung. Im Vergleich zu DS bietet iPOJO mehr Möglichkeiten. Deshalb wird im Rahmen dieser Arbeit iPOJO verwendet.

Kapitel 4

Analyse

In diesem Kapitel wird das Anwendungsszenario mit Hilfe von Anwendungsfällen dargestellt. Weiterhin werden die nicht funktionalen Anforderungen dargestellt.

4.1 Anwendungsszenario

Es wird ein CMS betrachtet, welches in einer Redaktion zur Verwaltung einer Internet-basierten Nachrichtenseite verwendet wird. In einem solchen Redaktionssystem wird der Content in Form von Dokumenten produziert und verwaltetet. Darunter sind auch Audio oder Videodokumente sowie einzelne Bestandteile eines Artikels, wie Bilderstrecken oder Absatzbilder zu verstehen. Wie in Abschnitt 2.1 erläutert, definiert ein CMS ein Statuskonzept für Dokumente. Dieses kann abhängig von dem Arbeitsablauf der Redaktion komplex sein.

Dokument-Status	Icon	Beschreibung
In Arbeit		Das Dokument wurde geändert und gespeichert. Das Dokument wird derzeit in einer vorherigen Version ausgespielt.
Released		(Optionaler Zustand) Explizit fertiggestelltes Dokument. Ein Zustand, der für Workflows eine Rolle spielen kann, z.B. für das Vier-Augen-Prinzip. Das Dokument wird derzeit in einer vorherigen Version ausgespielt.
Live		Das Dokument wird derzeit in genau dieser Version ausgespielt.
In Arbeit, Offline		Das Dokument wurde offline gestellt und seitdem verändert oder umgekehrt. Das Dokument wird nicht mehr ausgespielt.
Released, Offline		Das Dokument wurde offline gestellt und dann verändert und fertiggestellt. Das Dokument wird nicht mehr ausgespielt.
Live, Offline		Das Dokument wurde offline gestellt und seitdem nicht verändert.
Released, Jetzt keine Liveversion		(Optionaler Zustand) Explizit fertiggestelltes, noch nie ausgespieltes Dokument.
In Arbeit, Jetzt keine Liveversion		Das Dokument wurde geändert und gespeichert und noch nie ausgespielt.

Abbildung 4.1: Statuskonzept des CMS Sophora.

Abbildung 4.1 zeigt die Zustände eines Dokuments in dem CMS Sophora. Darüber hinaus existieren wohl definierte Übergänge von einem Zustand in einen anderen. Die meisten Status-Übergänge werden aktiv von einem Redakteur initiiert und gehören semantisch zu dem Arbeitsablauf. Bei anderen Übergängen wie z.B. von dem Zustand Released in den Zustand Live ist es sinnvoll einen Zeitpunkt zu definieren an dem dieser stattfinden soll.
Weiterhin werden Dokumente in einem Strukturbaum verortet. Dies entspricht der Einteilung der Nachrichtenseite in Kategorien, wie z.B. *sport/regional/fussball* oder *nachrichten/ausland/devisen*. Der Wurzelknoten dieses Baums ist die Nachrichtenseite an sich. Diese Struktur bildet einen Baum an dem die Dokumente die Blätter sind. Ein Strukturknoten entspricht einer Kategorie. Nun ist es sinnvoll auch Zeitplanungsdaten in den Strukturknoten angeben zu können. Diese vererben sich dann an alle Kindknoten und an alle konkreten Dokumente in den Blättern. Bei einer solchen kann es sich aber nicht um einen absoluten Zeitpunkt sondern nur um eine relative Zeitangabe handeln. Diese wird in Tagen nach dem Veröffentlichungszeitpunkt angegeben. So ist es möglich, beispielsweise alle Artikel unterhalb von *sport/regional/fussball* nach einer beliebigen Anzahl von Tagen offline zu stellen. Weiterhin ist es in der Praxis wichtig, diese generellen Einstellungen aus einem Strukturknoten in konkreten Dokumenten überschreiben zu können.
Überdies besteht die Anforderung bei Statusübergängen weitere Aktionen ausführen zu können. Beispielsweise kann die Notwendigkeit bestehen vor dem Veröffentlichen eines Audio- oder Videodokuments eine Konvertierung durchzuführen.
Im weiteren Verlauf wird die Komponente entworfen, welche die Zeitplanungsdaten über-

wacht, ausführt und in der Lage ist bei Statusübergängen weitere Aktionen auszuführen. Ein von dieser Komponente initiierter Statusübergang wird im folgenden Event genannt und die Komponente Eventserver

4.1.1 Anwendungsfälle

In diesem Abschnitt werden Anwendungsfälle dargestellt. Sie umfassen die Verwendung des Eventservers als Teil der Verwaltungskomponente.
Die Anwendungsfälle sind aus der Perspektive des CMS und nicht aus der Perspektive des Eventservers entwickelt wurden. Aus diesem Grund wird der Eventserver innerhalb der Anwendungsfälle als Akteur betrachtet.

Erstellen und veröffentlichen eines Artikels

Name	01: Erstellen und Veröffentlichen eines Artikels.
Erfolgsszenario	Ein Artikel wird durch einen Redakteur in der Erfassungskomponente erstellt. Zunächst hat der Artikel den Status Offline.
Beteilige Akteure	Redakteur, Erfassungskomponente, Eventserver, Verwaltungskomponente.
Verwendete Anwendungsfälle	Zeitgesteuertes veröffentlichen eines Artikels.
Auslöser	Nutzung des CMS zur Erstellung von Content.
Vorbedingungen	Das JCR ist erreichbar.
Nachbedingung	Die Daten wurden korrekt in dem Repository gespeichert.
Standardablauf	Der Artikel wird mit Zeitsteuerungsdaten angelegt. Der Eventserver ist dafür verantwortlich den Status des Artikels entsprechend den Zeitplanungsdaten zu steuern. Nach dem Veröffentlichen ist der Artikel durch die Ausspielungskomponente verfügbar.

Zeitgesteuertes veröffentlichen eines Artikels

Name	02: Zeitgesteuertes veröffentlichen eines Artikels.
Erfolgsszenario	Ein Artikel wird durch die Angabe eines Zeitpunkts von dem Eventserver veröffentlicht.
Beteilige Akteure	Erfassungskomponente, Eventserver, Verwaltungskomponente.
Verwendete Anwendungsfälle	Keine
Auslöser	Der entsprechende Artikel soll erst ab einem bestimmten Zeitpunkt ausgespielt werden.
Vorbedingungen	• Der Artikel wurde mit einem „Online ab“ Datum angelegt. • Das JCR ist erreichbar. • Die Datenbank des Eventservers wurde durch den Observer (siehe 5.3) korrekt mit dem Repository synchronisiert. • Das Publikationsdatum wurde erreicht.
Nachbedingung	Das Property isOnline des entsprechenden Artikels hat den Wert *true*.
Standardablauf	Siehe Abschnitt 5.5.

Steuern der Online-Verweildauer der Dokumente eines Strukturknotens

Name	03: Steuern der Online-Verweildauer der Dokumente eines Strukturknotens
Erfolgsszenario	Nach Ablauf der Zeitspanne, relativ zu dem Veröffentlichungsdatum eines jeden Artikels, sind betreffende Artikel unterhalb des Strukturknotens in dem die Eigenschaft „Tage bis Offline“ gesetzt ist, in dem Status Offline. Falls Artikel selbst Zeitplanungsdaten definieren müssen diese Werte die vererbten Werte des Strukturknotens überschreiben und dürfen nicht Offline gestellt werden.
Beteilige Akteure	Erfassungskomponente, Eventserver, Verwaltungskomponente.
Verwendete Anwendungsfälle	Erstellen und veröffentlichen eines Artikels.
Auslöser	Alle Dokumente eines Strukturknotens sollen eine definierte Zeit veröffentlicht sein.
Vorbedingungen	• Die Verweildauer wurde an dem Strukturknoten durch eine Angabe in dem Feld „Tage bis Offline“ definiert. • Das JCR ist erreichbar. • Die Datenbank des Eventservers wurde durch den Observer (siehe 5.3) korrekt mit dem Repository synchronisiert. • Die Verweildauer relativ zu dem Veröffentlichungsdatum der entsprechenden Artikel ist abgelaufen.
Nachbedingung	Das Property isOnline der entsprechenden Artikel haben den Wert *false*. Alle übrigen Artikel haben den gleichen Status wie zuvor.
Standardablauf	Das Event wird von dem Observer registriert, dieser aktualisiert die Datenbank. Der RelativeEventRunner stellt den Event fest und teilt dem Bundle EventHandler alle auszulösenden Events mit. Ab diesem Punkt wird das Event gleichbehandelt, wie absolute Events, die dem EventHandler gemeldet werden. Siehe auch 5.5

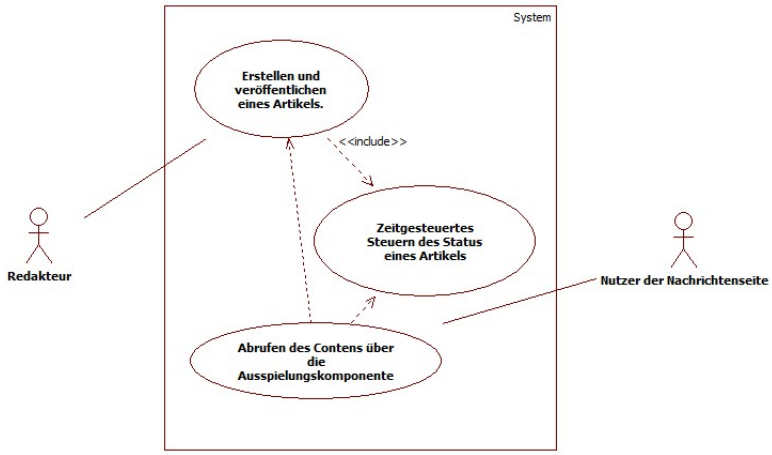

Abbildung 4.2: Anwendungsfalldiagramm.

Abrufen des Contents über die Ausspielungskomponente

Name	04: Abrufen des Contens über die Ausspielungskomponente
Erfolgsszenario	Ein Benutzer ruft die Nachrichtenseite auf.
Beteilige Akteure	Ausspielungskomponente, Nutzer, Verwaltungskomponente.
Verwendete Anwendungsfälle	Keine
Auslöser	Abruf der aktuelle Artikel durch den Benutzer.
Vorbedingungen	• Das JCR ist erreichbar. • Die Ausspielungskomponente ist erreichbar.
Nachbedingung	Die Artikel werden entsprechend ihres Status ausgespielt.

4.2 Nicht funktionale Anforderungen

Neben den funktionalen Anforderungen, die in Form von Anwendungsfällen dargestellt wurden existieren weitere nicht funktionale Anforderungen.

Zuverlässigkeit und Korrektheit: Definierte Zeitplanungsdaten müssen korrekt und zuverlässig werden. Ist dies nicht der Fall, können z.B. bereits erstellte und freigegebene Artikel nicht veröffentlicht werden. Zudem werden Artikel nicht mehr Offline gestellt, das hat zur Folge, dass die Internetpräsenz schnell in einen veralteten Zustand gerät und verbleibt. Dies kann dann nur noch manuell, durch direktes Setzten der Status

behoben werden. Korrektheit in diesem Kontext bedeutet, das ausschließlich Statusübergänge von dem Eventserver initiiert werden, die von Redakteuren durch entsprechende Konfiguration gewünscht sind.

Handhabung: Die Konfiguration des Eventservers muss möglichst einfach und intuitiv erfolgen. Auch Administratoren, die in der Regel nicht an der Entwicklung der von ihnen administrierten Systeme beteiligt sind und so kein Detail-wissen haben, müssen den Eventserver konfigurieren können.

Effizienz: Eine zeitliche Häufung von Events oder eine zeitlich Häufung von definierten Zeitplanungsdaten darf keine relevante Verzögerung nach sich ziehen. Alle Events müssen zeitnah ausgeführt werden. Zeitnah ist hier relativ zu dem Überprüfungsintervall zu sehen, in welchem überprüft wird ob Events ausgeführt werden müssen. Konkret bedeutet dies weniger als 60 Sekunden. Dies ist besonders für den Online-Journalismus entscheidend. Dieser lebt von seiner Schnelligkeit und der zeitnahen Berichterstattung.

Wartbarkeit: Die einzelnen Bundles sowie deren Zusammenspiel muss einfach wartbar sein, damit auftretende Fehler in einer akzeptablen Zeit gefunden und gelöst werden können. Auch fördert eine gute Wartbarkeit die Umsetzung von neuen oder geänderten Anforderungen.

Sicherheitsanforderungen: Es darf nicht möglich sein für unautorisierte Benutzer Zeitplanungsdaten zu verändern oder einzusehen. Dies könnte z.B. ein Veröffentlichen eines noch nicht freigegeben Artikels nach sich ziehen. Oder das Offline stellen aller veröffentlichten Artikel.

Kapitel 5

Entwurf

Der Entwurf beinhaltet die Beschreibung eines Prototyps, der als Rahmen für die Komponente dient, die technischen Anforderungen und den technischen Entwurf der Komponente.

5.1 System Szenario

Aus den in 4.1 beschriebenen Anforderungen wird insofern abstrahiert, dass Zeitplanungsdaten nur an den Dokumenten definiert werden können. Weiterhin existieren pro Dokument nur die beiden Status Online bzw. Offline und Archiviert bzw. nicht Archiviert. Die möglichen Übergänge sind in dem Statusdiagramm in Abbildung 5.1 ersichtlich.

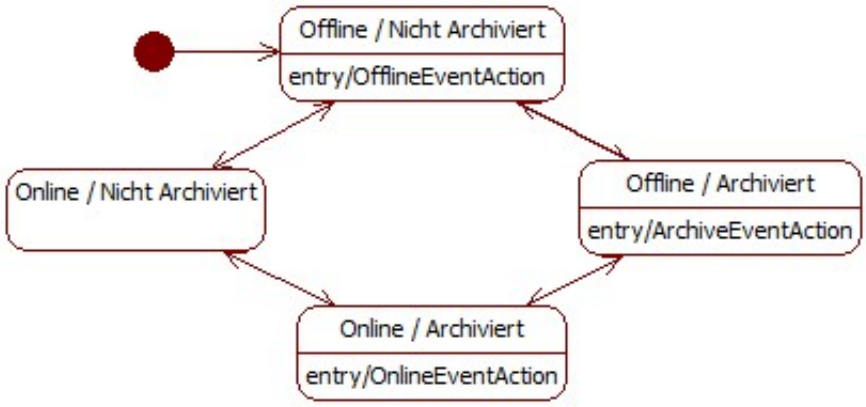

Abbildung 5.1: Statusübergänge

Bei der Aktion Archivieren wird der Artikel mit einer speziellen Kennzeichnung ausgespielt. Das Bezugsdatum für die relativen Events ist das Publikationsdatum. Dieses definiert

das Datum an dem ein Artikel veröffentlicht wird. Es wird entweder von der Erfassungskomponente gesetzt, wenn der Artikel direkt veröffentlicht oder von dem Eventserver, wenn der Artikel durch diesen veröffentlicht wird. Wenn ein Artikel entweder von der Erfassungskomponente oder von dem Eventserver Offline gestellt wird, wird das Publikationsdatum gelöscht. Um die Events zu registrieren wird eine Polling-Strategie verwendet. Hierbei kommen zwei unterschiedliche Polling-Strategien zum Einsatz. Bei den relativen Events reicht eine tägliche Überprüfung aus, bei den absoluten ist eine Überprüfung in einem beliebigen sekündliche Intervall ratsam. Diese Intervalle sollen von außen konfigurierbar sein. Um die Last auf das JCR durch sekündliche Überprüfungen zu vermeiden, hat der Eventserver eine interne Datenrepräsentation über die Zeitplanungsdaten. Weiterhin ist es möglich Abhängig von dem Typ eines Artikel bestimmte Aktionen beim Auftreten eines Events auszuführen. Ein Dokument definiert folgende Zeitplanungsdaten:

Name	Bedeutung	Aktion	Zeitangabe
Online ab	Das Datum, ab dem der Artikel veröffentlicht wird.	Veröffentlichen	Absolut
Online bis	Bis zu diesem Datum, bleibt der Artikel veröffentlicht.	Offline stellen	Absolut
Tage bis archiviert	Anzahl der Tage ab dem Publikationsdatum, nach denen der Artikel archiviert wird.	Archivieren	Relativ
Tage bis offline	Anzahl der Tage ab dem Publikationsdatum, nach denen der Artikel offline gestellt wird.	Offline stellen	Relativ

Tabelle 5.1: Zeitplanungsdaten.

Der Eventserver ist in der Verwaltungskomponente eines CMS angesiedelt. Abbildung 5.2zeigt die Einordnung durch eine Konkretisierung der in Abbildung 2.2 bereits verwendeten Darstellung.

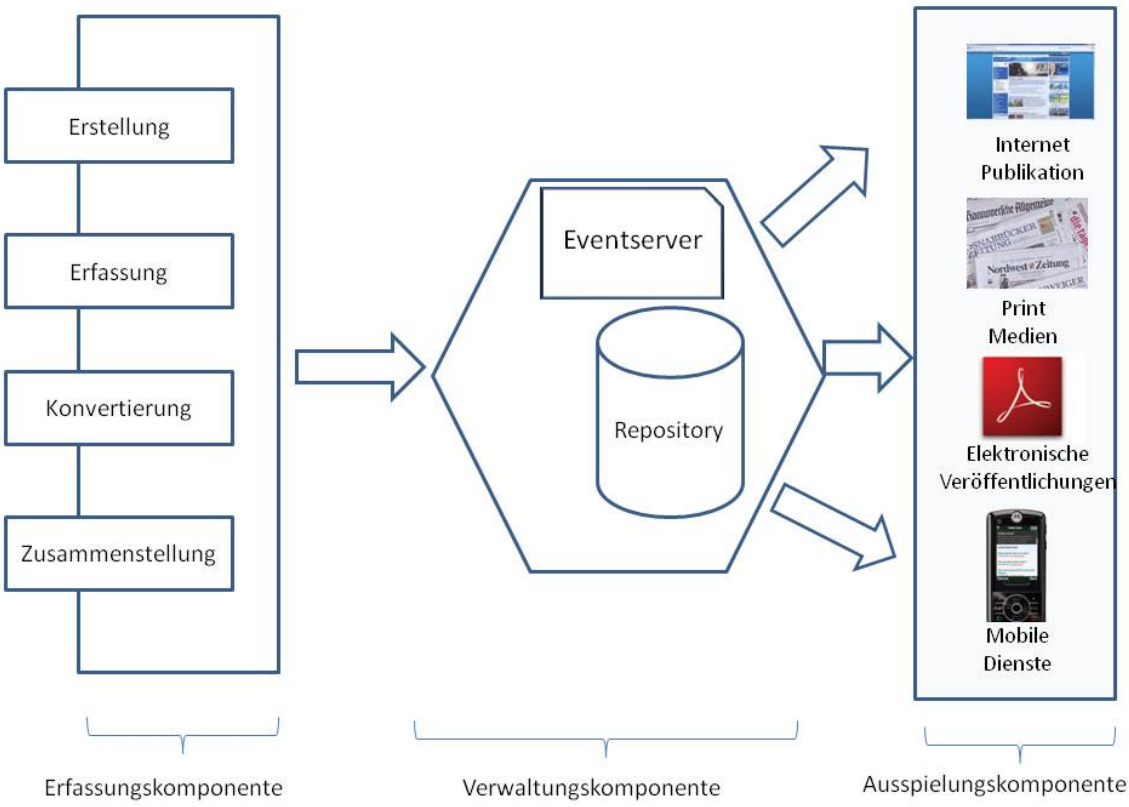

Abbildung 5.2: Einordnung des Eventservers.

5.2 Prototyp

Um die Funktionsfähigkeit des Eventservers herzustellen, wird ein CMS durch einen explorativen Prototyp simuliert. Dieser beinhaltet eine Erfassungs- und eine Ausspielungskomponente. Daneben ist es notwendig ein Node Type Modell zu erstellen, welches die Daten hält. Auf dieses Modell ist sowohl die Erfassungs - als auch die Ausspielungskomponente zugeschnitten. Beide arbeiten direkt auf dem JCR.

5.2.1 Node Type Modell

Um Einheitlichkeit bei der Vergabe von Namen zu erreichen, werden folgende Namensräume vergeben:

Namensraum	Verwendung
kuche-content-nt	Für node type Definitionen.
kuche-content	Für properties von nodes.
kuche-content-mix	Für mixins.

Tabelle 5.2: Namensräume.

Die Zeitplanungsdaten werden in dem mixin *timedates* zusammengefasst. So ist es möglich diese Gruppe von properties bei Bedarf weiteren Dokumenttypen hinzuzufügen. Für Artikel wird eine node type *story* angelegt. Seine Eigenschaften wie der Artikeltext oder das Publikationsdatum werden in properties abgelegt. Es wird durch das mixin *timedates* erweitert. Die Teaserdaten werden in dem childnode *teaser* zusammengefasst. Sowohl der Teaser als auch der Artikel node type erben von der gemeinsamen Oberklasse *ContentBase*. Diese wird durch das mixin *mix:referenceable* erweitert. Es existiert ein Wurzelknoten für den Content, den Typ Site. Dieser kann beliebig viele Kindelemente des Typs *ContentBase* enthalten. Die von dem mixin *mix:referenceable* zur Verfügung gestellte UUID, wird nicht dazu verwendet um Referenzen aufzubauen, sondern dazu um einen Artikel oder einen Teaser eindeutig zu identifizieren. Die Struktur der Node Types wird in Abbildung 5.3 verdeutlicht.

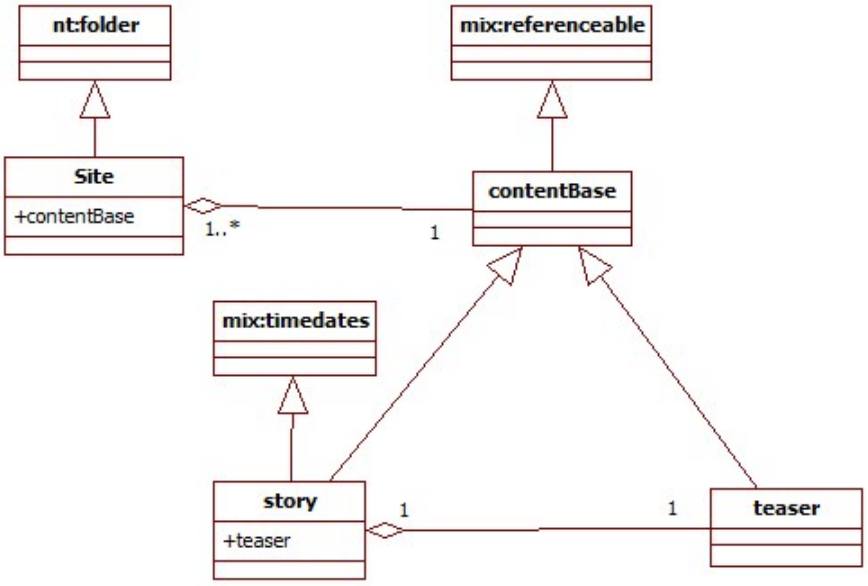

Abbildung 5.3: Struktur der Node Types.

5.2.2 Erfassungskomponente

Diese Komponente ist in der Lage eine Verbindung mit dem JCR herzustellen. Es wird eine Übersicht der bereits angelegten Artikel benötigt und es muss möglich sein neue anzulegen. Alle properties eines Artikel müssen in einer Detailansicht einsehbar und manipulierbar sein. Es ist notwendig Artikel löschen zu können.

5.2.3 Ausspielungskomponente

Die Ausspielungskomponente stellt einzelne Artikel entsprechend ihres Status auf einer Übersichtsseite dar. Es werden nur Artikel angezeigt, welche veröffentlicht sind. Archivierte Artikel müssen gesondert dargestellt werden. Über einen Link auf der Übersichtsseite ist die Detailansicht eines jeden Artikels abrufbar.

5.3 Design der Bundles

Die Unterteilung einer Applikation in einzelne Komponenten ist eine grundlegende Design Entscheidung, die die Entwicklung, den Test, die Wartung und den Betrieb der Applikation beeinflussen. Die folgenden Komponenten wurden so geschnitten, dass sie einen möglichst hohen Zusammenhalt und eine geringe Kopplung aufweisen. Wobei ein größerer Augenmerk auf den Zusammenhalt gelegt wurde. Auch weil dadurch mehr Bundles entstehen und dies praxis-nahen Umständen entspricht.

Zu beachten ist, dass die API eines jeden Service anbietenden Bundles in dem Bundle enthalten ist. Üblicherweise besteht die Notwendigkeit, die API zu kapseln und unabhängig von einer Implementation zur Verfügung zu stellen. Diese Notwendigkeit wird besonders deutlich, wenn mehrere unabhängige Teams beteiligt sind und diese sich nur über die Schnittstelle verständigen. Im Rahmen dieser Arbeit besteht diese Anforderung nicht und eine Umsetzung verspricht auch keine theoretischen Erkenntnisse.

DataBaseProvider ist für die Bereitstellung einer Datenbank zuständig. Diese Komponente kapselt eine Datenbank, die dadurch austauschbar gemacht wird.

DBManager bietet die Schnittstelle zur Kommunikation mit der Datenbank an. Die Realisierung der Datenbankschnittstelle, das Speichern und Auslesen von Datensätzen ist eine Verantworlichkeit. Diese wird von dem DBManager erfüllt.

Observer synchronisiert die Datenbank mit dem Repository. Wenn durch die Erfassungskomponente Artikel angelegt oder manipuliert werden, aktualisiert der Observer die Datenbank.

RelativeEventRunner überprüft einmal täglich ob Events ausgelöst werden müssen, welche durch eine relative Zeitangabe angegeben sind. Dies betrifft die beiden Felder „Tage bis archiviert“ und „Tage bis Offline“.

FixedEventRunner überprüft regelmäßig in einem definierbaren Zeitintervall ob ein Event ausgelöst werden muss, welcher durch eine absolute Zeitangabe angegeben ist. Dies betrifft die beiden Felder „Online ab“ und „Online bis“.

EventHandler Nimmt die Events der Komponenten RelativeEventRunner und FixedEventRunner entgegen. Abhängig von dem Typ des Artikels werden hier weitere Aktionen ausgeführt. Der EventHandler manipuliert dementsprechend die Datenbank und das Repository.

RepositoryManager stellt eine Schnittstelle zu dem Repository dar. Die Kommunikation zwischen EventHandler und Repository wird über diese abgewickelt.

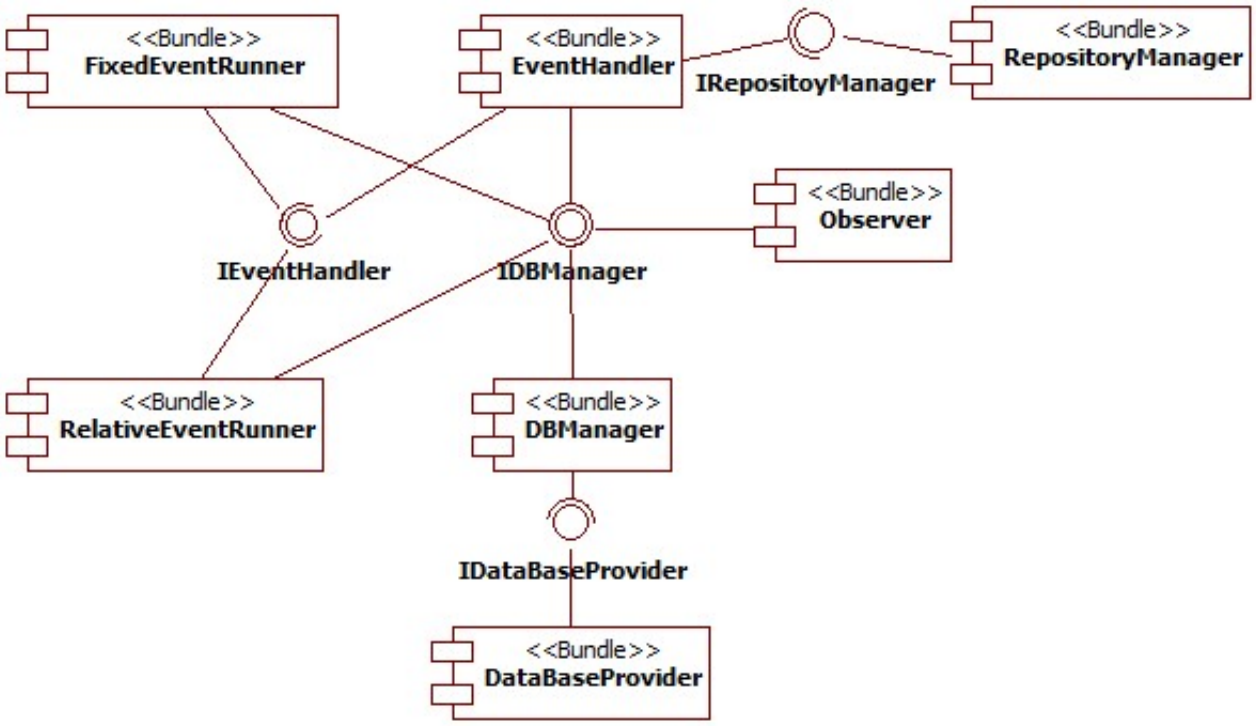

Abbildung 5.4: Komponentendiagramm der Bundles.

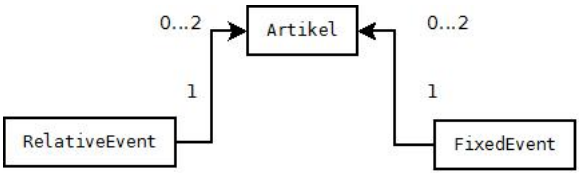

Abbildung 5.5: ERM des Datenbank Schemas.

5.4 Detailbeschreibung der Bundles

In den folgenden Abschnitten werden die einzelnen Komponenten im Detail dargestellt.

5.4.1 DataBaseProvider

Dieses Bundle stellt eine Datenbank zur Verfügung und ist für das Starten und Stoppen dieser verantwortlich. Beim Starten der Datenbank wird überprüft, ob das Datenbank Schema bereits angelegt wurde. Ist dies nicht der Fall wird es angelegt. Dies ist für den erst Start des Bundles wichtig. Das Datenbank Schema wird in dem Entity Relationship Model in Abbildung 5.5 veranschaulicht. Die 0...2 zu 1 Beziehung ergibt sich dadurch, dass sowohl für absolute als auch für relative Events zwei Event Typen existieren. Ein Event gehört immer zu genau einem Artikel und ein Artikel kann sowohl null bis zwei absolute und relative Events haben. Es wird ein Service unter dem Interface IDataSource zur Verfügung gestellt. Dieser Service nutzt die Service Strategie *Singleton.* Das bedeutet, dass alle Service Verbraucher eine Referenz auf eine Instanz des Service Anbieters erhalten. Dies ist nötig, sobald mehrere Service Verbraucher diesen Service nutzen, ansonsten würde jeder Service Verbraucher auf einer separaten Datenbank arbeiten.

5.4.2 DBManager

Dieses Bundle bietet eine Schnittstelle unter dem Namen IDBManager zur Datenbank an. Diese erfüllt die Anforderungen der Bundles Eventhandlers, FixedEventRunnder, Observer und RelativeEventRunner. Hier wird auch Anwendungslogik realisiert. So wird beim Löschen eines Events überprüft, ob der von dem Event referenzierte Artikel noch weitere Events referenziert. Ist dies nicht der Fall, besteht keine Notwendigkeit mehr den Artikel Datensatz zu speichern und wird gelöscht. Der Datenbankzugriff erfolgt durch den O/R Mapper Hibernate. Das package, welches die hierfür erforderlichen Mapping Klassen enthält wird exportiert und so den übrigen Bundles zur Verfügung gestellt.

5.4.3 Observer

Um die Datenbank bei Veränderungen durch die Erfassungskomponente mit dem Repository zu synchronisieren, observiert dieses Bundle alle Kindknoten des Nodes */news*. Dieses Bundle stellt keinen Service zur Verfügung. Es benötigt aber den Service IDBManager.

5.4.4 FixedEventRunner

Der FixedEventRunner ist dafür zuständig, die Events zu erkennen, welche durch ein absolutes Datum angegeben sind. Dies betrifft die Felder „Online ab“ und „Online bis“. Die sekündliche Überprüfung erfolgt über die IDBManager Schnittstelle. Beim Registrieren von Events werden die Events über die IEventHandler Schnittstelle an den EventHandler gemeldet. Weiterhin besteht über einen ManagedService die Möglichkeit das Überprüfungs-Intervall zu konfigurieren.

5.4.5 RelativeEventRunner

Der RelativeEventRunner arbeitet wie der FixedEventRunner nur mit dem Unterschied, dass dieser nur einmal täglich, zu einer festgelegten Uhrzeit startet um zu überprüfen, ob Events ausgelöst werden müssen, die auf einer relativen Zeitangabe basieren. Dies betrifft die Felder „Archivieren nach“ und „Offline nach“. Auch hier ist über einen ManagedService konfigurierbar zu welcher Uhrzeit die Überprüfung starten soll.

5.4.6 EventHandler

Der EventHandler nimmt die von den Bundles FixedEventRunner und RelativeEventRunner registrierten Events entgegen und verarbeitet diese. Zu diesem Zweck kommuniziert er über die Schnittstelle IDBManager mit der Datenbank und über die Schnittstelle IRepositoryManager mit dem Repository. Dieses Bundle stellt seinen Service unter der Schnittstelle IEventHandler zur Verfügung. Es ist in der Lage bei Statusübergängen von Artikeln, abhängig von dem Artikeltyp, weitere Aktionen auszuführen. Zu diesem Zweck werden in diesem Bundle drei Interfaces für die Aktionen Veröffentlichen, Offline stellen und Archivieren definiert. Dieses sind :

- IOnOnlineEventAction
- IOnOfflineEventAction
- IOnArchiveEventAction

Jedes dieser Interfaces schreibt eine Methode *run* vor, welche als Parameter eine Instanz der Klasse EventContext erhält. Aus diesem Parameter können Kontext Informationen des

Events ausgelesen werden. Abhängig von dem Artikeltyp werden dann in den implementierenden Klasse der Interfaces weitere Aktionen ausgeführt. Der EventHandler versucht Services, welche unter den hier beschriebenen Interfaces registriert sind zu binden. Gelingt dies nicht wird eine Default Implementation verwendet. Das Bundle EventHandler stellt einen Service unter der Schnittstelle IOnOnlineEventAction zur Verfügung. Dieser Service versendet Benachrichtigungs E-Mails, wenn es sich um den Artikeltyp „Nachricht“ handelt. Dadurch, dass für jeden dieser Services Standard-Implementation existieren, kann die Funktionalität bei Statusübergängen zur Laufzeit leicht erweitert oder wieder entfernt werden.

5.4.7 RepositoryManager

Der RepositoryManager nimmt über die Schnittstelle IRepositoryManager Events vom EventHandler entgegen und manipuliert dementsprechend das Repository. Der geleistete Service steht unter der Schnittstelle IRepositoryManager zur Verfügung. Weiterhin bietet das Bundle ein ServiceProperty an, über welches Verbraucher dieses Services erfahren können ob der Service eine Verbindung zum JCR hergestellt hat. Beim Binden dieses Services kann somit eine Filterung auf diesem ServiceProperty durchgeführt werden, sodass nur solche Services gebunden werden, welche mit dem JCR verbunden sind.

5.5 Interaktion der Komponenten

Der Ablauf vom Registrieren bis zum Ausführen eines Events wird durch das Sequenzdiagramm in Abbildung 5.6 veranschaulicht.

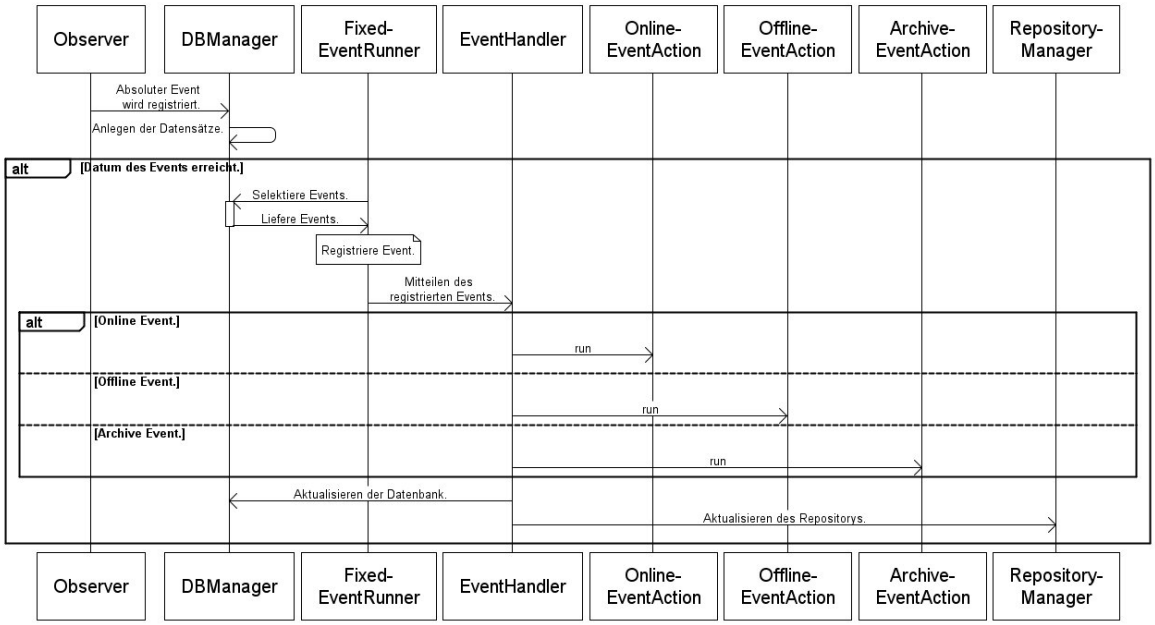

Abbildung 5.6: Sequenzdiagramm.

Zunächst wird ein Event durch den Observer registriert. In diesem Beispiel handelt es sich um einen durch eine absolute Zeitangabe definierten Event. Der Observer trägt nun über die IDBManager Schnittstelle entsprechende Datensätze in der Datenbank ein. Wenn die Zeitangabe erreicht ist, wird der Event durch den FixedEventRunner registriert. Dieser leitet die Information an den EventHandler weiter. Abhängig von dem Typ des Events wird nun die run Methode des entsprechenden Action-Handlers aufgerufen. Anschließend wird von dem EventHandler über die Schnittstellen IDBManager und IRepositoryManager die Datenbank und das Repository aktualisiert. Um Inkonsistenzen zu vermeiden muss das Aktualisieren der Datenbank und das Aktualisieren des Repositorys transaktional erfolgen.

5.6 Abhängigkeiten der Komponenten

Zur Laufzeit werden von den Bundles packages importiert und exportiert. Dies beschreibt explizit die Abhängigkeiten untereinander.

DataBaseProvider

Import	-
Export	org.kuche.bachelor.databaseprovider.api

DBManager

Import	org.kuche.bachelor.databaseprovider.api
Export	org.kuche.bachelor.dbmanager.api, org.kuche.bachelor.dbmanager.mappings

Observer

Import	org.kuche.bachelor.dbmanager.api, org.kuche.bachelor.dbmanager.mappings
Export	-

FixedEventRunner

Import	org.kuche.bachelor.dbmanager.api, org.kuche.bachelor.dbmanager.mappings, org.kuche.bachelor.repositorymanager.api, org.kuche.bachelor.eventhandler.api
Export	-

RelativeEventRunner

Import	org.kuche.bachelor.dbmanager.api, org.kuche.bachelor.dbmanager.mappings, org.kuche.bachelor.repositorymanager.api, org.kuche.bachelor.eventhandler.api
Export	-

EventHandler

Import	org.kuche.bachelor.dbmanager.api, org.kuche.bachelor.dbmanager.mappings, org.kuche.bachelor.repositorymanager.api, org.kuche.bachelor.eventhandler.api
Export	org.kuche.bachelor.eventhandler.api

RepositoryManager

Import	-
Export	org.kuche.bachelor.repositorymanager.api

5.7 Logging

In einer OSGi Anwendung ist das Planen einer Logging Architektur anspruchsvoller. Verwendete Fremd-Bibliotheken und einzelne Bundles einer OSGi Anwendung können unterschiedliche Logging Frameworks einsetzen. Dies wird besonders deutlich wenn man bedenkt, dass zur Entwicklungszeit unter Umständen noch nicht alle Bundles bekannt sind, die eingesetzt werden sollen. Die Schwierigkeit besteht nun darin unterschiedliche Logging Frameworks in einer OSGi Anwendung zusammenzubringen um diese einheitlich zu nutzten.
In der Java Umgebung existieren mehrere Logging Frameworks. Diese sind:

- Apache Commons - Logging (JCL)[21]
- JDK - Logging (JUL) [22]
- Log4J [23]
- SLF4J [24]

Alle vorgestellten Logging Frameworks werden an einer zentralen Stelle konfiguriert, das Logging an sich, also der programmatische Aufruf unterscheidet sich in OSGi Anwendungen nicht von dem in Standard Java Anwendungen.
Auch OSGi bietet eine eigene Lösung für das Logging an. Diese wird im Abschnitt 5.7.1 dargestellt.

SLF4J

SLF4J hat für den vorgestellten Lösungsansatz eine besondere Bedeutung. Aus diesem Grund wird es gesondert behandelt. SLF4J steht für „Simple Logging Facade for Java“. Es stellt eine Java Logging API zur Verfügung um das Entwurfsmuster der Fassade zu realisieren.
Das Entwurfsmuster Fassade gehört zu der Kategorie der Strukturmuster und bietet eine einheitliche Schnittstelle zu mehreren Schnittstellen eines Subsystems. Dadurch müssen die

[21] http://commons.apache.org/logging/
[22] http://java.sun.com/javase/6/docs/api/java/util/logging/package-summary.html
[23] http://logging.apache.org/log4j/
[24] http://www.slf4j.org/

Nutzer des Subsystem nicht alle Schnittstellen kennen, sondern lediglich dessen Fassade. Die Fassade delegiert die Aufrufe an das Subsystem. Durch diese Fassade ist das komplette Subsystem nutzbar (vgl. (GHJV04), S. 212). Mit SLF4J ist es möglich das verwendete Logging Framework zur Laufzeit auszutauschen. Darüber hinaus existieren Brücken, die zwischen unterschiedlichen Logging Frameworks vermitteln können.

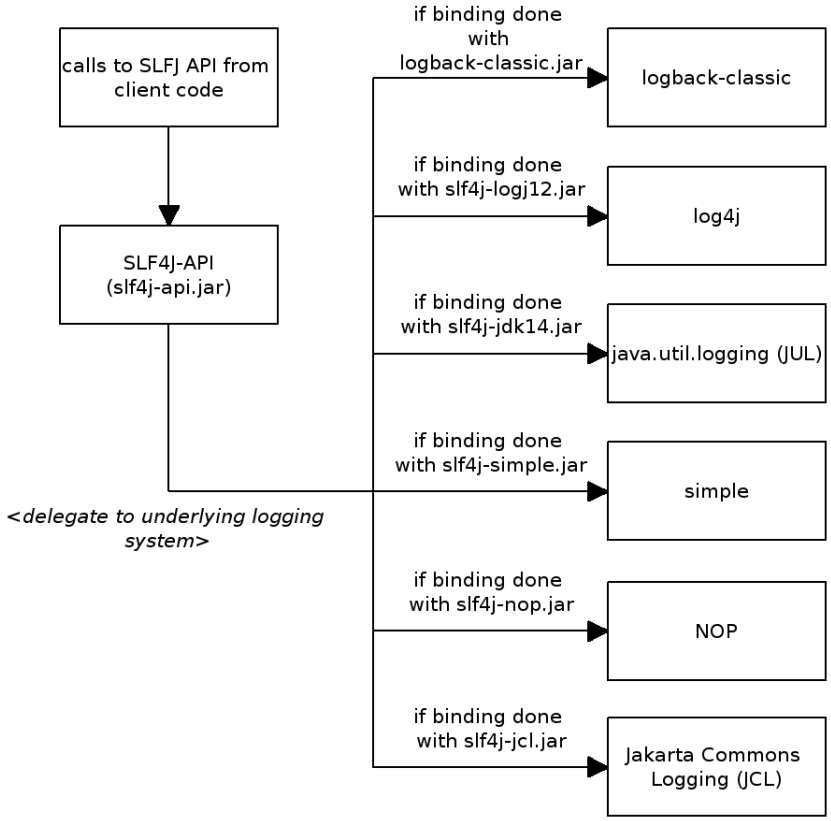

Abbildung 5.7: SLF4J als Fassade (SLF).

5.7.1 OSGi Log Service

Ähnlich wie die klassischen Logging Frameworks arbeitet der OSGi Log Service mit unterschiedlichen Log-Leveln. Dadurch kann man eine Log-Nachrichten klassifizieren, z.B. kann eine Nachricht nur von Relevanz sein, wenn ein Fehler gesucht wird. In diesem Fall erhält die Log-Nachricht den Level Debug. In der Konfiguration kann dann konfiguriert werden, welche Level momentan ausgegeben werden sollen.
Das Besondere an dem OSGi Log Service ist, dass dieser aus einem *LogService*, *LogReadern* und *LogListenern* besteht. Die Log-Nachrichten werden von dem *LogService* entgegengenommen. Über die *LogReader* können alle bereits stattgefundenen Log-Nachrichten

ausgelesen werden. Außerdem besteht die Möglichkeit sich durch einen *LogListener* an dem LogReader anzumelden und dann über künftige Log-Nachrichten benachrichtig zu werden. (vgl. (WHKL08), S. 289)
Beispielsweise können Management Agents *LogListener* nutzen um die Log-Nachrichten an einer zentralen Stelle, z.B einem grafischen Fromtend, anzuzeigen. Der *LogService* ist sowohl Bestandteil des OSGi Service Compendium als auch der OSGi Mobile Specification. Klassische Logging Frameworks erzeugen immer dann Log-Nachrichten wenn das entsprechende Level der Log-Nachricht konfiguriert ist. Der OSGi Logging Service geht hier anders vor. Es wird immer eine Log-Nachricht erzeugt, über die *LogReader* kann dann entschieden werden welche Ereignissen ausgelesen werden. Außerdem kann beim Erzeugen einer Log-Nachricht eine Service Reference mitgegeben werden, damit den Verbrauchern der Log-Nachricht dessen Quelle bekannt ist.
Um den OSGi Log Service unter Apache Felix zu verwenden muss zunächst das Bundle „Apache Felix Log Service“ installiert und gestartet werden. Um ihn zu nutzten wird eine Abhängigkeit zu dem Service *LogService* definiert.

5.7.2 Integration der Logging Frameworks

Um die unterschiedlichen Logging Frameworks zu integrieren wird SLF4J in Verbindung mit LOGBack verwendet. LOGBack ist eine native SLF4J Implementation und ein Teil von SLF4J. So kann es ohne Brücke von SLF4J verwendet werden. Es werden alle Log-Nachrichten über SLF4J Brücken zu dem Logging Framework LOGBack umgeleitet. Da nicht alle Logging Frameworks die gleichen Log Level besitzen ist eine Zuordnung der Level auf LOGBack über die Brücken notwendig. LOGBack ist dann für die Ausgabe der Log-Nachricht verantwortlich und abhängig von dessen Konfiguration werden die Log-Nachrichten aller Logging-Frameworks in einem oder mehreren Ausgabekanälen ausgegeben.

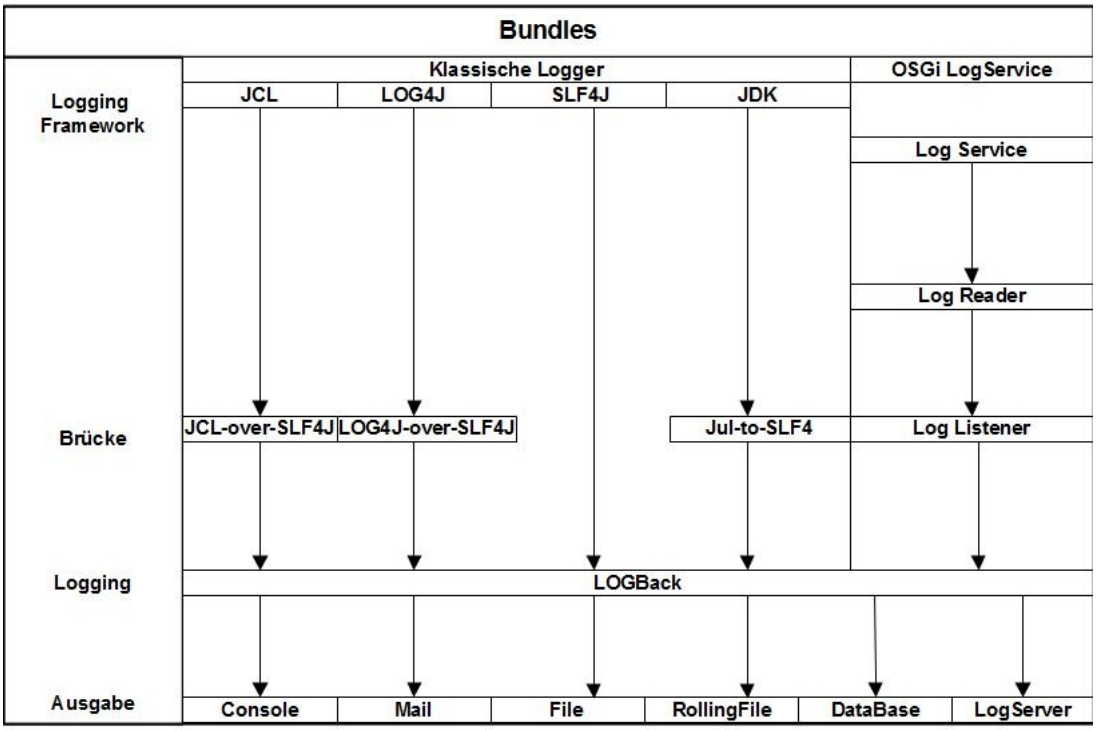

Abbildung 5.8: Integration der Logging Frameworks. (angelehnt an (Gen08))

Fazit

Für den Eventserver bietet sich die Nutzung des OSGi Log Service an. Bei diesem Ansatz ist die höchste Flexibilität gegeben. Wie erläutert kann auch dieser mit allen anderen integriert werden und darüber hinaus können Management Agents oder interessierte Bundles die Log-Nachrichten durch Log Reader erhalten.

Kapitel 6

Implementierung

6.1 Prototyp

Für die Veranschaulichung der Funktionalität des Eventservers wird ein explorativer Prototyp eines Content Management Systems benötigt. Dieser beinhaltet eine Erfassungskomponente und eine Ausspielungskomponente für die Internet Publikation.

6.1.1 Node Type Modell

Die Definition der Node Types erfolgt in der in Abschnitt 2.2.4.2 erläuterten CND Notation.

```
<'kuche-content-nt'='http://www.kuche.de/kuche-content-nt/1.0'>
<'kuche-content'='http://www.kuche.de/kuche-content/1.0'>
<'kuche-content-mix'='http://www.kuche.de/kuche-content-mix/1.0'>
<'nt'='http://www.jcp.org/jcr/nt/1.0'>
<'mix'='http://www.jcp.org/jcr/mix/1.0'>

['kuche-content-nt:site'] > nt:folder, nt:base
+ 'kuche-content-nt:contentBase' ('kuche-content-nt:contentBase')
    multiple

['kuche-content-nt:contentBase'] >  mix:referenceable,nt:base

['kuche-content-nt:teaser'] > 'kuche-content-nt:contentBase'
orderable
- 'kuche-content:shorttext' (string)
- 'kuche-content:headline' (string)

['kuche-content-nt:story'] > 'kuche-content-nt:contentBase',
                        'kuche-content-mix:timedates'
orderable
- 'kuche-content:publicationDate' (date)
- 'kuche-content:text' (string)
- 'kuche-content:online' (boolean)
- 'kuche-content:archiviert' (boolean)
- 'kuche-content:storyTyp' (string)
+ 'kuche-content:teaser' ('kuche-content-nt:teaser')

['kuche-content-mix:timedates'] mixin
-'kuche-content:onlineAb' (date)
-'kuche-content:onlineBis' (date)
-'kuche-content:offlineNach' (long)
-'kuche-content:archiviertNach' (long)
```

Abbildung 6.1: Daten Modell.

6.1.2 Erfassungskomponente

Für diese Komponente wurde eine Java Applikation mit einer grafischen Oberfläche nach dem „Model View Controller“ Entwurfsmuster erstellt. Die Kommunikation mit dem JCR wird über RMI realisiert und geht über die voreingestellte Adresse `rmi://localhost:1099/jackrabbit`. Die Verbindung wird durch den Menüpunkt „Connect“ hergestellt. Anschließend werden in der linken Spalte alle verfügbaren Artikel angezeigt. Durch das Selektieren eines Artikels werden in der rechten Spalte die einzelnen Eigenschaften zur Bearbeitung angezeigt. Nach dem Bearbeiten muss der „Speichern“ Button in der oberen Menüleiste ausgewählt werden um die Änderung dauerhaft zu speichern. In dieser Menüleiste sind auch die Funktionen „Neuer Artikel anlegen“ und „Artikel löschen“ zu finden. In der rechten oberen Ecke der Menüleiste wird ein Icon zur Veranschaulichung des Verbindungsstaus angezeigt. Ebenfalls ist hier ein Link zu finden, der die Ausspielungskomponente in einem Browser öffnet. Für den Eventserver relevante Daten sind die Zeitplanungsdaten und die beiden Felder Online bzw. Offline und Archiviert. Abbildung 6.2 stellt eine Ansicht der Erfassungskomponente dar.

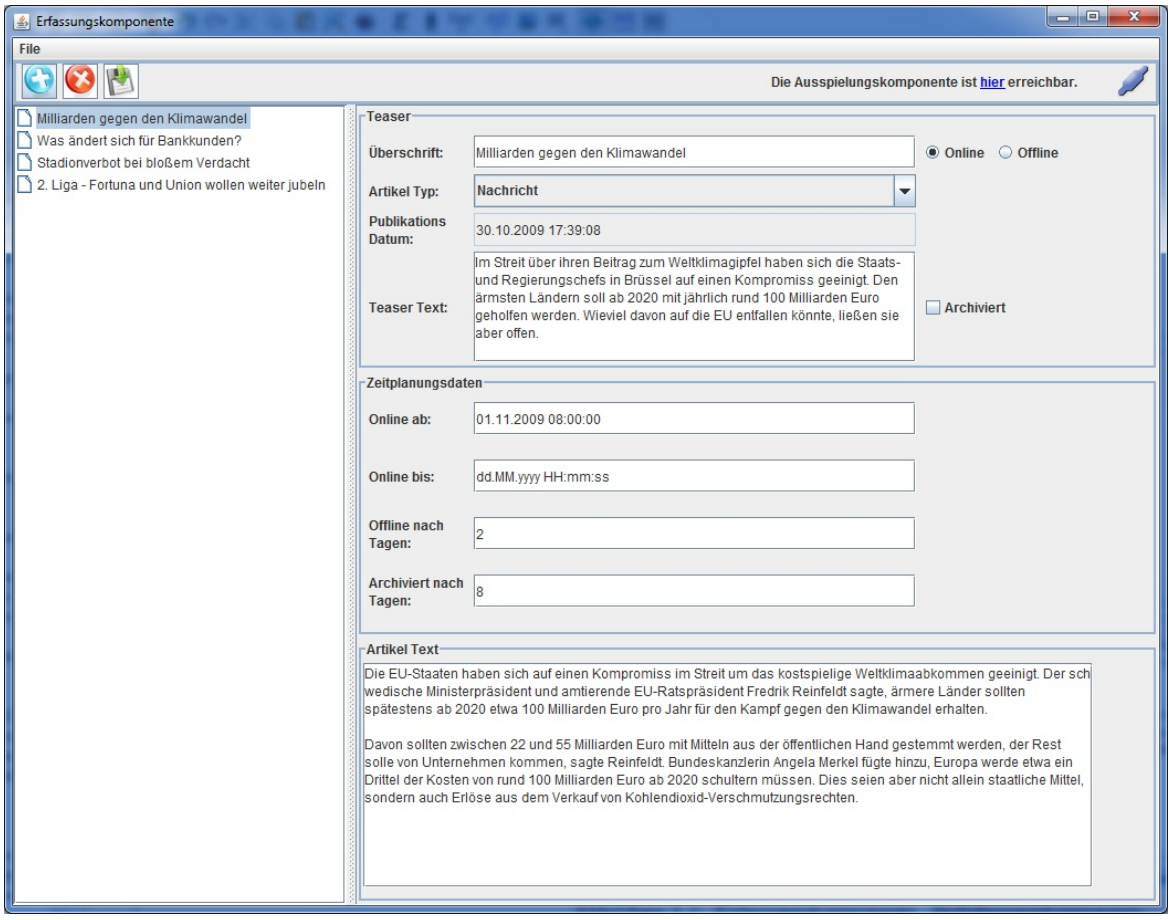

Abbildung 6.2: Erfassungskomponente.

6.1.3 Ausspielungskomponente

Die Ausspielungskomponente besteht im Wesentlichen aus zwei Elementen. Zum einen ist dies eine Übersichtsseite über alle im JCR gespeicherten Artikel und zum anderen eine Detail-Ansicht für jeden Artikel. Die Übersichtsseite ist unter der URL `http://localhost:8080/news.html` erreichbar. Zur Veranschaulichung des Status Archiviert, wird der Hintergrund eines archivierten Artikels in der Übersichtsseite gesondert dargestellt. Abbildung 6.3 veranschaulicht die Übersichtsseite der Ausspielungskomponente für die in Abschnitt 6.2 dargestellten Artikel. Der Artikel mit der Überschrift „Was ändert sich für Bankkunden“ ist archiviert und wird deswegen grau hinterlegt dargestellt. Der Artikel „Stadionverbot bei bloßen Verdacht“ hat den Status Offline und wird in der Ausspielungskomponente nicht dargestellt. Diese Komponente wurde mit Hilfe von Apache Sling realisiert. Apache Sling wird in Abschnitt 2.4 vorgestellt. Zum Auslesen und zum Darstellen des Contents wird Javascript und die Javascript Bibliothek `sling.js` verwendet.

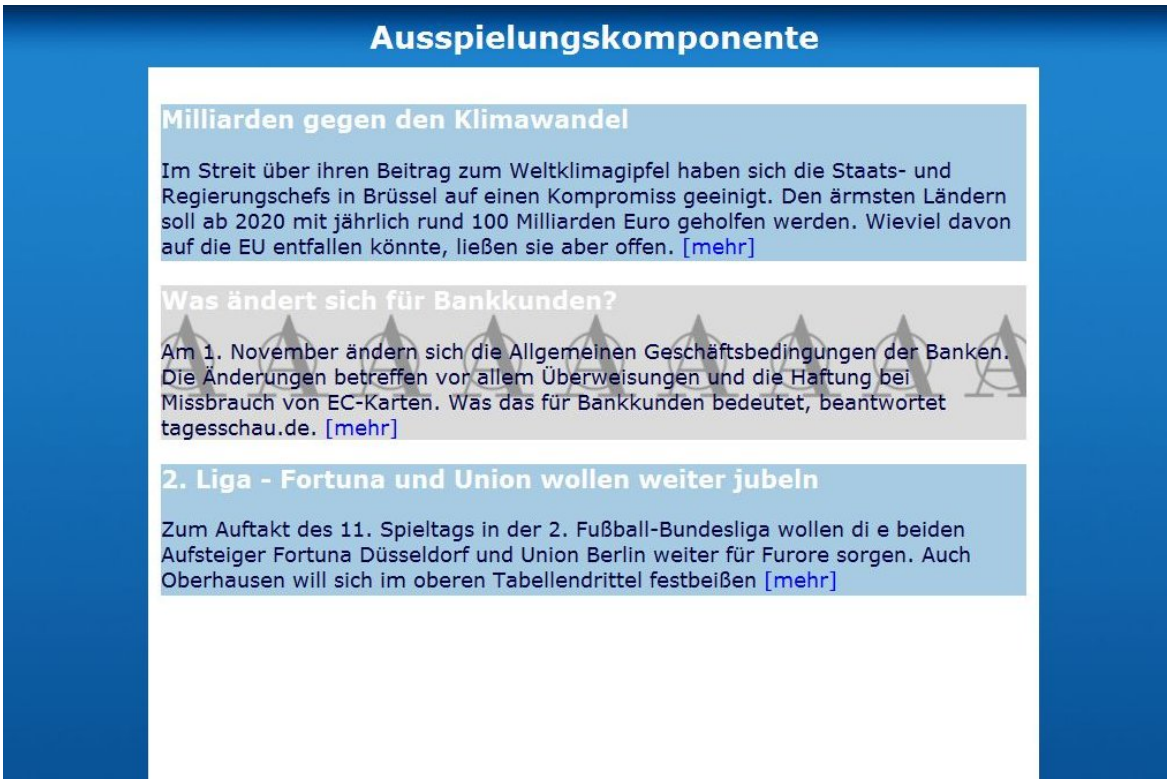

Abbildung 6.3: Ausspielungskomponente.

6.2 Integration von Apache Felix in Eclipse

Um die Entwicklung von Applikationen auf Grundlage der OSGi Service Plattform in der Praxis praktikabel zu machen ist es zum einen notwendig einen Build Prozess für die Applikation zu erstellen. Dies ist in Abschnitt 6.3.1 beschrieben. Zum anderen ist es nötig den Komfort und die Unterstützung von modernen IDEs bei der Entwicklung nicht zu verlieren. Dazu gehören das Debuggen von Code und das Ausführen von Code aus der Entwicklungsumgebung heraus. Um diese beiden Anforderungen umzusetzen ist es notwendig eine OSGi Implementation in die Entwicklungsumgebung zu integrieren. Bei Apache Felix existieren zu diesem Zweck prinzipiell drei Möglichkeiten:

1. Durch die Nutzung von Pax Runner[25].
2. Integration einer Felix Version durch ein Java Projekt.
3. Integration des Felix trunks als Java Projekt.

[25] `http://paxrunner.ops4j.org/space/Pax+Runner`

Hier wird die zweite Möglichkeit verwendet. Dazu wird ein neues Java Projekt in der Entwicklungsumgebung angelegt und die Dateien einer Felix Version in dieses kopiert. Daraufhin kann ein target angelegt werden, welches Felix starten und debuggen kann. Um die Entwicklung weiter zu vereinfachen, sind einige Konfigurationen in der Datei `config.properties` notwendig. Durch das property *felix.auto.start.1* können alle Bundles angegeben werden, die beim Starten im start level 1 gestartet werden sollen. Typischerweise werden hier die Pfade eingetragen, die das Build Werkzeug als Ausgabe verwendet. Es ist auch möglich ein einzelnes Bundle zu aktualisieren nachdem die OSGi Service Plattform bereits gestartet wurde. Dies leistet der Befehl *update*<BundleId>. Werden packages dieses Bundles durch weitere importiert, ist darüber hinaus ein *refresh* notwendig. Hierdurch werden die Importe aktualisiert.

6.3 Implementierung der Bundles

Die Entwicklung der Bundles erfolgt in der IDE Eclipse. Die OSGi spezifischen Elemente werden im Zuge der Implementierung mit iPOJO realisiert. Für das Konfigurationsmanagement wird Maven 2 verwendet. Um Maven in Eclipse zu integrieren kommt das Eclipse Plugin „Maven for Eclipse integration“ zum Einsatz.

6.3.1 Das Konfigurationsmanagement mit Maven 2

Laut einer Studie der Lawrence Livermore National Lab werden in komplexen Softwareprojekten 12-35% der Entwicklungszeit für den Build-Prozess verwendet. Dazu gehören die Auseinandersetzung mit dem Build-Tool, das Warten auf langsame Builds oder das Suchen nach Phantom-Bugs aufgrund von inkonsistenten Builds. (vgl. (Kum02), S. 17).
Besonders im Kontext von OSGi ist es wichtig einen effizienten Build-Prozess zu etablieren. Das Hauptargument für die Verwendungen einer Software für das Konfigurationsmanagement im Zusammenhang mit OSGi, ist das automatische Erzeugen von Bundles. Das händische Editieren der `MANIFEST.MF` Datei ist sehr fehleranfällig und zeitaufwändig. Prinzipiell kann für diese Aufgabe ANT oder Maven verwendet werden. Aufgrund der einfachen Erzeugung von Projekten durch die Verwendung eines Archetypes und die Möglichkeit die Abhängigkeiten zur Compile-Zeit zu verwalten wird im Folgenden Maven verwendet.
Für jedes Bundle wird in Eclipse ein Java Projekt erstellt. Die Projekte werden durch den Maven Archetype *maven-ipojo-plugin* erstellt. Dadurch wird direkt beim Erstellen eines Projektes eine übliche Ordnerhierarchie angelegt. Wie in Abschnitt 6.3.1 beschrieben werden mit Hilfe von Maven die Abhängigkeiten der Projekte zur Compile-Zeit definiert. Die notwendige Konfiguration des Build-Prozesses findet in der Konfigurationsdatei `pom.xml` statt. Um Maven im Kontext eines OSGi Projekts unter Verwendung von iPOJO zu nutzen sind die beiden Maven Plugins *maven-bundle-plugin* und *maven-ipojo-plugin* notwendig.

Konfiguration eines Projekts durch die pom.xml

Die Datei `pom.xml` ist die zentrale Konfigurationsdatei in Maven. In ihr werden die Abhängigkeiten eines Projekts definiert. Maven greift zur Auflösung dieser Abhängigkeiten auf Repositories zu. Diese sind nicht mit jenen aus Abschnitt 2.2 zu verwechseln. Hierbei handelt es sich um lokale oder entfernte Verzeichnisse. Innerhalb dieser sind eine Vielzahl von freien Bibliotheken verfügbar. Die Bibliotheken werden alle durch die Angabe einer Abhängigkeit und deren Version aus diesem Verzeichnis geladen und dem Build Path des Projekts hinzugefügt. Eine Abhängigkeit beschreibt sich hierbei immer eindeutig durch ihre groupid, artifactid und ihrer Versionsnummer. Ferner wird in der pom.xml die groupid, die artifactid und die Versionsnummer des Projekts festgelegt.
Auch die Build-Informationen sind hier zu finden. Aus diesen wird dann die `MANIFEST.MF` Datei innerhalb der resultierenden jar Datei erzeugt. In folgendem Ausschnitt wird eine verkürzte Form der pom.xml des Projekts Eventhandler dargestellt:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <packaging>bundle</packaging>
  <groupId>org.kuche.bachelor</groupId>
  <artifactId>EventHandler</artifactId>
  <version>0.1</version>
  <name>EventHandler</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.apache.felix.ipojo.annotations</artifactId>
      <version>1.2.0</version>
    </dependency>
    ...
  </dependencies>
  <build>
    <plugins>
      <plugin>
      <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>1.4.3</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>
              ${pom.groupId}.${pom.artifactId}
            </Bundle-SymbolicName>
            <Private-Package>
              org.kuche.bachelor.eventhandler,
              org.kuche.bachelor.eventhandler.dummyactions,
              org.kuche.bachelor.eventhandler.actions
            </Private-Package>
            <Import-Package>
              org.kuche.bachelor.dbmanager.api
                ;version="0.1",
              org.kuche.bachelor.dbmanager.mappings
                ;version="0.1",
              org.kuche.bachelor.repositorymanager.api
                ;version=""[0.1, 1.0)",
              *;resolution:=optional
          </Import-Package>
            <Export-Package>
              org.kuche.bachelor.eventhandler.api
```

```
44                    ;version="${pom.version}"
45                </Export-Package>
46                <Embed-Dependency>
47                  *;artifactId=!DBManager|RepositoryManager
48                </Embed-Dependency>
49              </instructions>
50            </configuration>
51          </plugin>
52          <plugin>
53            <groupId>org.apache.felix</groupId>
54            <artifactId>maven-ipojo-plugin</artifactId>
55            <executions>
56              <execution>
57                <goals>
58                  <goal>ipojo-bundle</goal>
59                </goals>
60                <configuration>
61                  <metadata>
62                    src/main/resources/metadata.xml
63                  </metadata>
64                </configuration>
65              </execution>
66            </executions>
67        </plugins>
68    </build>
69 </project>
```

Für die Generierung der `MANIFEST.MF` Datei sind folgende Tags entscheidend:

Bundle-SymbolicName: Definiert den eindeutigen symbolischen Namen des Bundles.

Private Package: Definiert die packages, die in dem Bundle enthalten aber nicht exportiert werden sollen.

Import Package: Diese Abhängigkeiten werden zur Laufzeit importiert. Zu beachten ist die Deklaration **;resolution:=optional.* Dadurch wird gekennzeichnet, dass alle in den Java Klassen deklarierten Imports versucht werden zur Laufzeit aufzulösen. Diese sind jedoch optional. So ist das Bundle startbar, obwohl nicht alle Abhängigkeiten befriedigt sind. Die Angabe ist wichtig, falls mit Fremd-Bibliotheken gearbeitet wird. Oftmals bauen diese auf weiteren Bibliotheken auf und haben ein hohe Zahl an Abhängigkeiten. Wenn man aber nur ein Teil der angebotenen Funktionalität benötigt, müssen auch nicht alle Abhängigkeiten befriedigt werden.
Darüber hinaus kann innerhalb der Import Deklaration eines packages eine oder die

akzeptierten Versionen angegeben werden. Bei einem Bereich von akzeptierten Version wird die aus der Mathematik bekannte Intervall Schreibweise verwendet. Bei der Versionangabe in Zeile 39 handelt es sich um ein rechtsoffenes Intervall. Alle Versionen in dem Intervall 0.1,1.0 werden akzeptiert. Einschließlich 0.1 und ausschließlich 1.0.

Export Package: Dies ist die Liste der zu exportierenden Packages. Die Angabe einer Version ist möglich aber nicht zwingend. In diesem Fall wird über die Variable *pom.Version* die Version des Bundles verwendet.

Embed-Dependency: Hier werden die Abhängigkeiten definiert, welche in Form von jar Dateien dem resultierenden Bundle hinzugefügt werden. Entscheidend ist hierbei, dass die beiden Artefakte mit den artifactids DBManager und RepositoryManager ausgenommen werden. Diese beiden Artefakte sind als Abhängigkeit definiert, weil sie zur Compile-Zeit benötigt werden. Zur Laufzeit sollen diese Abhängigkeiten allerdings importiert werden.

6.3.2 Verwendung der iPOJO Annotationen

Die Deklarationen von OSGi spezifischen Mechanismen im Java Code erfolgen durch die Verwendung von iPOJO Annotationen. Im Folgenden werden diese anhand von Beispielen aus den Bundles Eventhandler und FixedEventRunner erläutert.

```
1 @Component(name="EventHandler")
2 @Provides
3 public class EventHandler implements IEventHandler{
4 ...
```

Durch die Annotation *@Component* in diesem Beispiel wird eine Factory mit dem Bezeichner EventHandler angelegt. Hierbei handelt es sich um eine Variante des Factory-Patterns. Die Factory ist in der Lage Instanzen der annotierten Klasse zu erzeugen. Die Annotation *@Provides* registriert einen Service unter den Namen aller implementierenden Interfaces. Dies entspricht Schritt 1 in Abbildung 2.9.

```
1  @Requires(defaultimplementation=DummyArchiveEventAction.class,
2  optional=true)
3  private IOnArchiveEventAction onArchive;
4
5  @Requires(optional=false,id="dbManger")
6  private IDBManager dbManager;
7
8  @Requires(optional=false,id="repoManger",
9  filter="(connected=true)")
10 private IRepositoryManager repoMan;
```

In Zeile 1 wird definiert, dass ein Service, der das Interface IOnArchiveEventAction implementiert, zur Laufzeit gebunden werden soll. Allerdings ist diese Abhängigkeit als optional gekennzeichnet und eine Standard Implementation ist angegeben. Hierbei handelt es sich bereits um dynamische Services. Falls zur Startzeit kein entsprechender Service zur Verfügung steht, wird zunächst die Standard Implementation verwendet. Wird zu einem späteren Zeitpunkt ein entsprechender Service registriert, wird dieser sofort gebunden. Weiterhin kann nach bestimmten Eigenschaften eines Services gefiltert werden. In den Zeilen 8 - 10 wird nur ein Service akzeptiert, welcher ein ServiceProperty unter dem Bezeichner *connected* definiert und dieses den Wert true hat. Auf diese Weise werden nur Services unter dem Interface IRepositoryManager gebunden, die eine Verbindung zu dem JCR hergestellt haben.

```
1  @Bind(id="dbManger")
2  private void setDbManger(IDBManager dbManger) {
3  ...
4
5  @Unbind(id="dbManager")
6  private void unBindDBManger(){
7  ...
8
9  @Validate
10 private void start(){
11 ...
12
13 @Invalidate
14 private void stop(){
15 ...
```

Es ist möglich durch die Deklarationen von Callback Methoden auf bestimmte Ereignisse, wie Zustandswechsel oder dem Binden und Freigeben von Abhängigkeiten zu reagieren.

```
1 @Component(name="FixedEventRunner",
2                 managedservice = "FixedEventRunnerMS")
3 public class FixedEventRunner {
4 ...
5
6 @Property(name = "intervall")
7 public void set(String intervall) {
8 ...
```

Durch die Deklarationen eines *managedservice* ist es möglich eine Komponente zur Laufzeit zu konfigurieren. Die Konfigurationen kann auf unterschiedliche Art und Weise geschehen. In diesem Fall wird das Bundle „Apache Felix File Install“[26] verwendet. Hierbei handelt es sich um ein auf Verzeichnissen basierendes Management Modell. Alle in ein bestimmtes Verzeichnis kopierten Bundles werden hierdurch automatisch installiert und gestartet. Werden die Bundles aus dem Verzeichnis gelöscht, werden sie deinstalliert. In dieses Verzeichnis können auch Konfigurationsdateien abgelegt werden. In dem oben beschriebenen Beispiel muss diese den Dateinamen *FixedEventRunnerMS. cfg* haben. Durch die Zeile *intervall=50000* innerhalb der Datei wird das Property Intervall auf den Wert 50000 gesetzt. Bei jeder Veränderung dieses Wertes wird die Methode *set(String intervall)* aufgerufen. Innerhalb dieser wird ein TimerTask neu konfiguriert und das Zeitintervall zur Überprüfung von Events neu gesetzt.

Bei optional markierten Service Abhängigkeiten gibt es eine Besonderheit zu beachten: iPOJO arbeitet mit dem Null Object Pattern. Falls kein Service Provider zur Laufzeit zur Verfügung steht, hat die entsprechende Instanzvariable keine Referenz auf *null* sondern auf ein „Null Objekt“. Das „Null Objekt“ erfüllt die Schnittstelle, führt aber keinerlei Aktionen aus. Damit ist es nicht nötig vor der Verwendung einer solchen Abhängigkeit auf *null* zu prüfen, um *NullPointerExceptions* zu vermeiden. Die Verwendung des Null Object Pattern kann aber auch durch den Parameter *nullable=false* in der *@Requires* Annotation ausgeschaltet werden. In diesem Fall ist aber die Überprüfung auf *null* vor einer Verwendung angebracht.

Instantiierung von iPOJO Instanzen

Durch die Verwendung von Annotationen ist es nicht möglich iPOJO Instanzen zu erzeugen. Diese werden dadurch erzeugt, dass sie innerhalb einer Datei *metadata.xml* definiert wer-

[26] http://felix.apache.org/site/apache-felix-file-install.html

den. Diese muss sich nicht zwangsläufig in dem gleichen Bundle befinden in welchem die entsprechende Factory durch *@Component* Annotation definiert wird.

```
1 <ipojo >
2         <instance component="EventHandler"
3                 name="EventHandlerInstance"/>
4         <instance component="OnlineEventAction"
5                 name="OnlineEventActionInstance"/>
6 </ipojo>
```

Abbildung 6.4: Instantiierung von iPOJO Instanzen.

6.4 Verwendete Bibliotheken

Im Folgenden werden verwendete Bibliotheken vorgestellten und evtl. wird ein Hinweis darauf gegeben wo und wie diese Verwendet werden.

H2 Die relationale Datenbank H2[27] ist in Java geschrieben hat einen sehr kleinen Footprint und wird bevorzugt in Anwendungen eingebettet. Generell bietet sie die beiden Möglichkeiten Tabellen im Arbeitsspeicher zu halten, oder persistent auf der Festplatte zu speichern. Weiterhin werden eine Vielzahl von Funktionalitäten angeboten wie, Transaktionen, Verschlüsselung oder Volltextsuche. Diese Datenbank wird in dem Bundle DataBaseProvider verwendet.

Hibernate Hibernate ist der am weitesten verbreitetste Object-Relational-Mapping Framework in Java. Es wird in dem Bundle DBManager verwendet. Durch die Verwendung von Hibernate ist es möglich Java Objekte in relationalen Datenbanken zu speichern und umgekehrt wieder aus entsprechenden Datensätzen auszulesen.

Apache Commons Lang Hierbei handelt es um eine Sammlung von Werkzeugen, die die Basisklassen in *java.util* erweitern.

Apache Commons IO Diese Bibliothek bietet Werkzeuge zur besseren Handhabung von Streams.

javax.Mail Stellt eine einfache Schnittstelle zum Versenden und Abrufen von E-Mails über mehrere Protokolle zur Verfügung.

[27] `http://www.h2database.com/html/main.html`

Kapitel 7

Test

Die Teststrategie beinhaltet OSGi-spezifische Testfälle und jUnit Tests für das Bundle DB-Manager.

7.1 OSGi spezifische Testfälle

Aus den Anwendungsfällen 01 und 02 in Abschnitt 4.1.1 werden Testfälle konstruiert. Hierbei handelt es sich um dynamische Blackbox Tests. Die Testfälle werden durch jUnit4OSGi Tests realisiert. Die Testfälle sollen auch die Frage klären inwieweit diese Anwendung skalierbar ist. Aus diesem Grund ist auch ein Lasttest enthalten. In diesem werden 1500 Testdaten generiert. Die Interaktion des Redakteurs über die Erfassungskomponente mit dem Repository wird durch die direkte Interaktion des Tests mit dem Repository simuliert.

7.1.1 jUnit4OSGi

Für die hier verwendeten Testfälle kommt das Testframework jUnit4OSGi zum Einsatz. jUnit4OSGi ist ein Unterprojekt von iPOJO und wird von der Apache Software Foundation entwickelt. Es erweitert jUnit in der Version 3.0 um Funktionalitäten für das Testen von serviceorientierten Anwendungen in einer OSGi Umgebung. Dabei werden die Testfälle innerhalb eines Bundles definiert, oder es wird ein extra Test Bundle erzeugt in dem nur die Testklassen angegeben sind. Die letztere Möglichkeit wird im Folgenden verwendet. Dazu wird ein Maven Bundle Projekt mit dem Namen „EventServer TestSuite“ angelegt. Die *pom.xml* dieses Projektes unterscheidet sich von jener in Abschnitt 6.3.1 nur dadurch, dass zusätzlich die Klassen angegeben werden die eine TestSuite darstellen. Dazu wird in dem tag, der durch den folgenden xPath Ausdruck

/project/build/plugins/plugin[artifactId='maven-bundle-plugin']/configuration/instruction

selektiert wird das tag *Test-Suite* hinzugefügt. In diesem werden durch eine Kommaseparierte Liste die entsprechenden Klassen oder package Namen angegeben.
Weiterhin ist es sinnvoll die korrekte Funktionsfähigkeit von Bundles oder einzelnen Services durch reguläre jUnit Tests zu gewährleisten. Dennnoch ist es notwendig die korrekte Interaktion der Bundles und Services zur Laufzeit sicherzustellen.
jUnit4OSGi übernimmt dabei die Konzepte von jUnit. So erweitert eine Testklasse nicht wie üblich die Klasse TestCase sondern die Klasse OSGiTestCase. Eine TestSuite wird durch eine Klasse realisiert, die eine Methode mit der Signatur

public static Test suite(BundleContext bc)

enthält. Die Klasse *OSGiTestCase* stellt den erbenden Klassen eine Reihe von Methode zur Verfügung, die eine Interaktion mit Services ermöglichen. Beispielsweise seien hier die folgenden genannt:

boolean isServiceAvailable(String svc)
boolean isServiceAvailableByPID(String itf, String pid)

Auch werden Methoden zur Verfügung gestellt, um Bundles zu installieren, zu starten oder zu stoppen. Es kann der *BundleContext* erfragt oder die package Importe aktualisiert werden.

Ausführung von jUnit4OSGi Testfällen

Um jUnit4OSGi Testfälle auszuführen, wird zunächst das Bundle mit den Testfällen installiert und gestartet. Die zu testenden Bundles müssen installiert und gestartet sein. Sofern sie nicht durch die Testfälle selbst installiert und gestartet werden. Außerdem muss die junit4OSGi runtime installiert sein. Diese wird durch das Bundle „org.apache.felix.ipojo.junit4osgi“ zur Verfügung gestellt. Um den Test auszuführen benötigt man TestRunner. Hierbei existieren unterschiedliche Möglichkeiten, die sich in der Art und Weise unterscheiden, wie die Tests ausgeführt und der Testbericht dargestellt wird. Z.B. stellt das Bundle „org.apache.felix.ipojo.junit4osgi.felix-command“ den zusätzlichen Befehl *junit* zur Verfügung. Er wird aufgerufen mit der Bundle-Id des Bundles dessen Tests ausgeführt werden sollen. Im Folgenden wird der Runner „org.apache.felix.ipojo.junit4osgi.swing-gui“ verwendet. Dieses stellt eine grafische Oberfläche zur Verfügung. Aus dieser heraus können alle in der OSGi Plattform enthaltenen Tests ausgeführt werden.

7.1.2 Spezifikation der Testumgebung

Das Repository muss erreichbar sein und die in Abschnitt 6.1.1 definierten node types müssen geladen sein. Der Eventserver muss vor dem Starten des Tests gestartet werden. Zu Beginn der Testfälle werden das Repository und die Datenbank in einen definierten Ausgangszustand überführt.

Verfügbare Services während des Test

Die Testfälle decken jeweils die Funktionsprüfung einer bestimmten Anzahl von Services und Bundles ab. Aus diesem Grund ist es notwendig vor der Ausführung bestimmte Services zu de-registrieren und nach der Ausführung wieder zu registrieren. In den folgenden Testfällen werden die Services de-registriert, welche bei Statusübergängen von Artikeln weitere Aktionen ausführen. So kann die Funktionalität des Eventservers unabhängig von diesen getestet werden. Hierfür wird das Controller Konzept von iPOJO verwendet.

Das Controller Konzept von iPOJO

Mit Hilfe des Controller Konzepts kann der Status von iPOJO Instanzen zur Laufzeit verändert werden. Hierzu wird eine boolesche Variable definiert und als Controller deklariert. Dies ist nur durch die Deklaration innerhalb der XML Konfiguration möglich. Durch Annotations wird dieses Konzept nicht abgedeckt. Durch den Wert der Variablen kann der Status gesteuert werden. Wird sie auf *true* gesetzt hat die Instanz den Status valide andernfalls den Status invalid.

7.1.3 Testfälle

Testfall 01

Testziel	Dieser Test überprüft, ob alle benötigten Services zur korrekten Ausführung des Eventservers korrekt gestartet und verfügbar sind.
Vorbedingungen	siehe 7.1.2 Spezifikation der Testumgebung.
Durchführung des Tests	Der Test wird durch den TestRunner org.apache.felix.ipojo.junit4osgi.swing-gui gestartet.
Eingabedaten	Ist der BundleContext nach dem Starten des Eventservers.
Spezifikation des erwarteten Ergebnisses	Folgende Services müssen verfügbar sein: IDataSource, IDBManager, IEventHandler, IRepositoryManager.
Nachbedingungen	-
Verwendete Prüfanweisung	Mittels der Methode *boolean isServiceAvailable(String svc)* wird für jeden benötigten Service ermittelt, ob dieser verfügbar ist.

Testfall 02

Testziel	Ein Artikel, der mit einem „Online ab“ Datum erstellt wird, soll ab diesem Datum veröffentlicht sein.
Vorbedingungen	siehe 7.1.2 Spezifikation der Testumgebung.
Durchführung des Tests	Der Test wird durch den TestRunner org.apache.felix.ipojo.junit4osgi.swing-gui gestartet.
Eingabedaten	Es wird ein Artikel in dem Zustand Offline / Nicht Archiviert angelegt. Dieser wird mit einem „Online ab“ Datum in einer Minute versehen.
Spezifikation des erwarteten Ergebnisses	Die Datenbank des Eventhandlers muss von dem Bundle Observer synchronisiert werden. Beim Erreichen des definierten „Online ab“ Datums muss ein entsprechender Event verschickt und die Datenbank sowie das Repository aktualisiert werden.
Nachbedingungen	Nachdem das „Online ab“ Datum erreicht ist, muss dieser Artikel veröffentlicht sein.
Verwendete Prüfanweisung	Es wird überprüft, ob das Property *kuche-content:online* nach Erreichen des „Online ab“ Datums den Wert *true* hat.

Testfall 03

Testziel	Dies ist ein Lasttest und stellt sicher, dass auch bei Belastung die Datenbank des Eventservers korrekt synchronisiert wird und alle Events korrekt ausgeführt werden.
Vorbedingungen	siehe 7.1.2 Spezifikation der Testumgebung. Während des Test müssen in dem EventHandler die Default Implementationen für die Statusübergänge gebunden sein. Das Intervall des FixedEventRunner darf nicht länger als 60 Sekunden betragen.
Durchführung des Tests	Der Test wird durch den TestRunner org.apache.felix.ipojo.junit4osgi.swing-gui gestartet. Es werden zunächst 1500 Artikel angelegt und 1000 Datensätze in die Datenbank geschrieben. Anschließend werden 500 Events ausgeführt. Mit ausreichend großen Zeitpuffern zwischen den Ereignissen Artikel anlegen und Event Datum und außerdem zwischen Events ausführen und Testüberprüfung kann dieser Test bis zu zehn Minuten dauern.
Eingabedaten	Es werden 500 Artikel vom Typ Nachricht in dem Zustand Offline / Nicht Archiviert angelegt. Das „Online ab“ Datum dieser Artikel wird mit einem zufälligen Zeitpunkt assoziiert, welches zwischen fünf und sechs Minuten in der Zukunft liegt. Es werden 500 Artikel vom Typ Nachruf in dem Zustand Offline / Nicht Archiviert angelegt. Für diese wird kein „Online ab“ Datum definiert. Es werden 500 Artikel vom Typ Nachricht in dem Zustand Offline / Nicht Archiviert angelegt. Es wird ein „Online ab“ Datum in einem Jahr eingetragen.
Spezifikation des erwarteten Ergebnisses	Die Datenbank des Eventservers muss von dem Bundle Observer synchronisiert werden. Beim Erreichen des definierten „Online ab“ Datums müssen entsprechender Events verschickt und die Datenbank sowie das Repository aktualisiert werden.
Nachbedingungen	Nach dem das „Online ab“ Datum erreicht ist, müssen genau 500 Artikel das property *isOnline* den Wert *true* haben. In der Datenbank des Eventservers müssen 500 Datensätze über die Artikel gespeichert sein, welche in einem Jahr veröffentlicht werden sollen.
Verwendete Prüfanweisung	Die Anzahl der veröffentlichten Artikel wird festgestellt. Die Anzahl der in der Datenbank gespeicherten Datensätze wird ermittelt.

7.2 jUnit Testfälle für das Bundle DBManager

Der Service IDBManger stellt einen zentralen Service dar. Er wird von vier weiteren Services verwendet. Aus diesem Grund liegt es nahe die Funktionalität der Implementation dieser Schnittstelle mit jUnit Testfällen abzudecken. Die Abhängigkeit zu einem Service unter der Spezifikation
org.kuche.bachelor.databaseprovider.api.IDataSource
wird in den Testfällen durch direktes Erzeugen der Abhängigkeit simuliert.

Die Schnittstelle IDBManager umfasst zehn Methoden. Für jede dieser Methode existiert min. ein Testfall. Aufgrund der Anzahl der Testfälle wird an dieser Stelle nicht auf jeden einzelnen eingegangen.

Kapitel 8

Bewertung

In diesem Kapitel werden die Ergebnisse und Erkenntnisse dieser Arbeit bewertet. Um auf die Praxis-Tauglichkeit des Eventservers einzugehen wird auch dessen Betrieb betrachtet. Der Entwurf wird bewertet und mit den Anforderungen sowie mit der Implementierung abgeglichen. Dies dient dem Ziel um zu untersuchen inwiefern sich die gestellten Anforderungen, vor dem speziellen fachlichen Hintergrund, mit OSGi realisieren lassen. Auch werden die nicht funktionalen Anforderung betrachten, um zu untersuchen ob sich in deren Realisierung Nachteile durch die Verwendung von OSGi ergeben. Durch diese Bewertungen wird die in der Zusammenfassung gegebene Antwort auf die ursprüngliche Frage, wie sinnvoll sich CMS Anforderungen mit OSGi realisieren lassen, motiviert.
Darüber hinaus wird Kritik geübt. Diese beinhaltet die Ambitionen von Java zur Integration eines Konzepts der Modularisierung, um mögliche Auswirkungen auf OSGi zu untersuchen.

8.1 Modularität in Java

Auch in Java gibt es eine Reihe von Bestrebungen, ein Modul System in den Sprachumfang aufzunehmen und die JVM entsprechend zu erweitern. Dazu zählen der JSR 277 „Java Module System“, der JSR 294 „Improved Modularity Support in the Java Programming Language“, das Projekt Jigsaw und der JSR 291 „Dynamic Component Support for Java“. Alle hier aufgeführten Bestrebungen werden momentan noch äußerst kontrovers diskutiert. Laut einer Ankündigung von OpenJDK vom 14.09.2009 wird in der nächsten Java Version 7 die voraussichtlich im zweiten Quartal 2010 erscheint, der JSR 294 und das Projekt Jigsaw enthalten sein (vgl. (JDK09)). Der JSR 294 hat aber seit dem 11.12.2009 den Status inactiv (vgl. (Buc09)). Auch ist der JSR 277 inaktiv. Weiterhin finden sich Online Quellen von beteiligten Entwicklern in denen das Scheitern des JSR 294 diskutiert wird (vgl.(Bar09b); (Ble09)). Es ist also fraglich ob in absehbarer Zeit praktikable modulare Erweiterungen Einzug in die Programmiersprache Java halten.

Projekt Jigsaw

Das Ziel des Projekts Jigsaw ist es, das JDK an sich zu modularisieren. Die angestrebten Verbesserungen sind die Größe des Downloads, eine verbesserte Startdauer und einen kleineren Speicherverbrauch. In diesem Zuge soll auch die Java Platform modularisiert werden. Diese würde es erlauben Anwendungen nur mit den Teilen der JVM zu installieren die tatsächlich benötigt werden. Welches Modularisierungssystem hier zum Einsatz kommt ist noch nicht abschließend geklärt. Mark Reinhold, der Leiter des Projekts, schreibt hierzu:

> „This effort will, of necessity, create a simple, low-level module system whose design will be focused narrowly upon the goal of modularizing the JDK. This module system will be available for developers to use in their own code, and will be fully supported by Sun, but it will not be an official part of the Java SE 7 Platform Specification and might not be supported by other SE 7 implementations.
>
> If and when a future version of the Java SE Platform includes a specific module system then Sun will provide a means to migrate Jigsaw modules up to that standard. In the meantime we'll actively seek ways in which to interoperate with other module systems, and in particular with OSGi.“ (Rei08)

JSR 294

Der JSR 294 erweitert das package Konzept von Java. So kann neben der package Deklaration auch ein Modul deklariert werden, in dem das package untergeordnet ist. Über Annotations kann ein Modul versioniert werden und ebenfalls über Annotations können versionierte Module importiert werden. Im Folgenden wird ein Beispiel dargestellt (vgl. (Mil09)). Diese Deklarationen erscheinen noch vor der package Deklaration innerhalb einer Java Klasse.

```
@Version("2.3")
@MainClass("org.foo.Foosball")
@ImportModules {
    @ImportModule(name="java.se.core", version="1.7+")
    @ImportModule(name="org.bar", version="1.0", reexport="true")
}
@ExportResources({org/foo/icons/**})
module org.foo;
```

8.2 Kritik

Generell entsteht durch die Anwendung einer Technologie auch ein Bedarf an spezifischem Wissen und Erfahrung. Bei OSGi ist dies nicht anders. Es erfordert Entwickler mit entspre-

chenden Kenntnissen.
Die Unterteilung einer Anwendung in Bundles und Services und Abhängigkeiten auf package Ebene ist sehr nutzbringend. Daraus entsteht aber auch Kosten in Form von gesteigerten Anforderungen an die Dokumentation. Eine Entwickler-Dokumentation in diesem Zusammenhang muss alle Abhängigkeiten explizit enthalten. Dies betrifft natürlich auch die Versionierung. Denn es ist auch mit OSGi möglich den Import und Export Mechanismus auf package Ebene zu unterlaufen, indem man z.B. alle Abhängigkeiten durch einen Wildcard Ausdruck importiert und alle packages exportiert. In einem solchen Fall hat man gegenüber dem flachen Suchraum des Class Loaders einer Standard Java Anwendungen nichts gewonnen.

Auch entsteht durch eine service- und komponentenorientierte Architektur nicht zwangsläufig eine saubere Anwendungsarchitektur. Die Reduzierung der Komplexität zur Entwicklungszeit beispielsweise bedingt durch die Wiederverwandbarkeit, die Versionierung oder der Kapselung wird durch einen höheren Aufwand während der Entwurfsphase eingekauft. Die Granularität der Komponenten, die bei der Unterteilung der Anwendung entstehen, ist eine wichtige und nur schwer zu korrigierende Entscheidung.
Die Komplexität einer großen monolithischen Anwendung besteht nicht nur in der Komplexität der einzelnen Klassen sondern vielmehr darin wie diese Klassen zusammenarbeiten, in der Aggregation dieser. Durch die vielen „Has-A“ Beziehungen ergibt sich ein Klassengeflecht aus dem ein lauffähiges Programm ensteht. Bei vielen kleinen Bundles kann dasselbe Problem auf einer anderen Ebene entstehen. Die Verflechtungen der Bundles wird so unübersichtlich, dass diese zu einer gesteigerten Komplexität und zu Unverständlichkeit beitragen. Darüberhinaus ist bei der Komposition von vielen kleinen Bundles ein exaktes Wissen über jedes einzelne notwendig.
Auf der anderen Seite besteht die Gefahr durch zu große Komponenten eine ähnlich hohe interne Komplexität zu erreichen als dies bei monolithischen Anwendungen der Fall ist.

Weiterhin können nicht alle Frameworks ohne weiteres in einer OSGi Umgebung verwendet werden. Z.B. sollte innerhalb dieser Arbeit das Job Scheduling Framework Quartz[28] zum Einsatz kommen. Es hat sich aber herausgestellt dass dieses innerhalb einer OSGi Umgebung nicht lauffähig ist. Auch werden an sich statische Frameworks nicht alleine dadurch dynamisch, dass sie innerhalb eines OSGi Bundles verwendet werden. Wird der O/R Mapper Hibernate in einem Bundle verwendet und es kommen zur Laufzeit neue Mapping Klassen hinzu muss die Hibernate SessionFactory neu konfiguriert und neu gestartet werden. Die Konfigurationen muss also zum Startpunkt bereits festliegen.
Letztendlich trifft man eine weitreichende Entscheidung, wenn in einem großen Software Projekt die Entscheidung während des Entwurfs auf OSGi fällt. Diese Entscheidung betrifft

[28] `http://www.quartz-scheduler.org/`

die gesamte Architektur der Software. Bei einer langlebigen Software ist es wichtig, dass auch OSGi nicht in naher Zukunft von anderen Technologien abgelöst wird. Wie in Abschnitt 8.1 dargestellt existieren mehrere parallele Ansätze zu OSGi. Diese sind aber momentan noch nicht annähernd so stabil und ausgereift wie OSGi.

8.3 Bewertung

Für den Betrieb ergeben sich durch die modulare Architektur eine Reihe von Vorteilen. Es können leicht neue Versionen von einzelnen Bundles eingespielt werden ohne den Eventserver stoppen zu müssen. Dies ist insbesondere wichtig damit keine Inkonsitenzen zwischen Repository und Datenbank des Eventservers entstehen. Die Redaktion muss so für ein Update nicht die Arbeit einstellen. Die Bundles Observer, DBManager und DataBaseProvider reichen aus um die Datenbank mit dem Repository synchron zu halten. Diese können ungestört weiterarbeiten, während andere Bundles aktualisiert werden.
Durch den Mechanismus der Standard-Implementationen für Statusübergangs-Aktionen, kann das Verhalten bei Statusübergängen zur Laufzeit beliebig verändert werden.

Auch die Anforderungen bezüglich der Relativen und Absoluten Zeitplanungsdaten konnte durch die Polling-Stategie umgesetzt werden. Die Konfiguration der Überprüfungsintervalle kann durch Managed Services in Verbindung mit dem Bundle „Apache Felix File Install“ komfortable zur Laufzeit umgesetzt werden. Prinzipiell besteht auch die Möglichkeit ähnlich zu einem Interrupt, Jobs zu jedem Statusübergang zu definieren, welche zu einem bestimmten Zeitpunkt ausgeführt werden. Aber auch in diesem Fall müssen die Jobs persistiert werden, damit nach einem Neustart der Applikation die Jobs nicht verloren gehen. Auch in diesem Fall ist also eine Datenbank notwendig.

In dem explorativen Prototyp wurde von der Anforderung abstrahiert, dass sich Zeitplanungsdaten in einer hierarschichen Struktur vererben. Aber auch diese Anforderung ist mit der hier vorgestellten Architektur handhabbar. Der Observer müsste die Zeitplanungsdaten eines Strukturknotens, einem Knoten in der Vererbungshierachie, wie die eines Dokuments interpretieren und in der Datenbank speichern. Beim Registrieren dieses Events müsste das Bundle RelativeEventRunner, erkennen dass es sich um einen Strukturknoten handelt und Events für alle untergeordneten Artikel an das Bundle EventHandler melden.

Die Synchronisation der Datenbank des Eventservers mit dem Repository erfolgte durch das Observationskonzept von Jackrabbit. Bei dem Lasttest hat sich gezeigt, dass dieses Konzept relativ langsam arbeitet. Wird z.B. eine Node angelegt, so wird der Observer über mehrere Events benachrichtigt. Für das Anlegen der Node und das Anlegen jeder Property. Zudem erhält man die Events in Form eines Objekt der Klasse *javax.jcr.observation.EventIterator.*

Man muss über diesen Iterator iterieren den zugrunde liegenden Event klassifizieren und entsprechende Aktionen ausführen. Dadurch hat man eine hohe Kopplung gegenüber der Struktur des Repositorys. Die Inperformanz dieses Ansatzes ist darin begründet, dass nun jeder Event ausgewertet und klassifiziert werden muss.
In einem produktiven System ist ein Listener Konzept angebracht. Dadurch könnte der Eventserver von der Verwaltungskomponente benachrichtigt werden, wenn für ihn interessante Veränderungen im Repository stattgefunden haben. Bei dieser Benachrichtigung könnte direkt eine leicht auszuwertende Datenstruktur übergeben werden.

Wie in dem Abschnitt 8.2 Kritik, aufgeführt besteht sowohl eine Gefahr darin die Bundles zu klein als auch zu umfangreich zu entwerfen. In der hier vorgestellen Architektur sind die Bundles zu klein geschnitten. In 5.3 wird die Designentscheidung motiviert. Auch in Hinblick auf den in Abschnitt 9.2 vorgeschlagen Schritt die komplette Verwaltungskomponente auf Basis der OSGi Service Plattform zu entwickeln, ist es denkbar das der komplette Eventserver als ein Bundle entworfen wird. Möchte man jedoch den Vorteil wahren, dass Aktualisierungen ausgeführt werden können ohne Inkonsistenzen zwischen Datenbank und Repository zu riskieren, sind mindestens zwei Bundles notwendig.

Hinsichtlich der nicht funktionalen Anforderungen, die in 4.2 dargestellt werden ergeben sich durch die Nutzung von OSGi keine Nachteile. Durch die modulare Architektur ergeben sich für die Anforderungen der Wartbarkeit und der Skalierbarkeit sogar Vorteile. Tritt ein Fehler auf und kann dieser einem Bundle zugeordnet werden beschränkt sich die Fehlersuche auf das Bundle. Neue Anforderungen können entweder durch weitere Bundles oder nur in der Modifikation einzelner Services oder Bundles bewerkstelligt werden. Dies fördert die Wartbarkeit und die Skalierbarkeit.

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst. Des Weiteren werden in Abschnitt 9.2 Themen und Aspekte dargestellt, die im Rahmen dieser Arbeit nicht behandelt wurden oder die diese sinnvoll erweitern.

9.1 Zusammenfassung und Fazit

Es hat sich gezeigt, dass sich die Anforderungen für diesen fachlichen Hintergrund gut mit OSGi abdecken lassen. In dem Entwurf wurde deutlich, dass bei der Unterteilung einer Anwendung in Services weitreichende Design Entscheidungen getroffen werden müssen. Dies trifft auch auf diesen Kontext zu. Deutlich wurde aber auch dass die guten Entwurfsprinzipien der Modularisierung und der serviceorientierten Architektur einen wirklichen Nutzen und Mehrwert für alle Phasen eines Softwareprojekts haben. Auch Content Management Systeme können davon profitieren. Im Rahmen dieser Arbeit wurde der Einsatz von OSGi innerhalb der Verwaltungskomponente diskutiert. Abhängig von der Form der Erfassungskomponente ist auch hier der Einsatz von OSGi denkbar, z.B. wenn die Erfassungskomponente als Webapplikation realisiert ist.
Bei der Entwicklung hat sich gezeigt, dass es eine Vielzahl von unterstützenden Werkzeugen gibt. Außerdem muss nicht auf den Komfort von modernen IDEs verzichtet werden. Das Konfigurationsmanagement eines OSGi Projekts wurde in diesem Rahmen mit Hilfe von Maven durchgeführt. Durch die gute Unterstützung ist das Konfigurationsmanagement praktikabel und nicht wesentlich aufwändiger als das anderer Softwareprojekte. Auch ist das Testen von OSGi Anwendungen praktikabel. Es wird durch entsprechende Technologie wie jUnit4OSGi oder den Integrationstests auf dem Spring DM Server unterstüzt. Weiterhin ist es möglich jUnit Tests innerhalb von Bundles einzusetzen und so die Funktionalität der einzelnen Bundles zu testen. Alles in allem ist OSGi eine ausgereifte Technologie, deren Einsatz auch im Rahmen von Content Management Systemen nichts im Wege steht.

9.2 Ausblick

In dieser Arbeit wurde innerhalb des explorativen Prototyps von der Verwaltungskomponente abstrahiert. Nur der zu dieser zugehörige Eventserver wurde betrachtet. Eine konsequente Fortführung ist es die komplette Verwaltungskomponente auf Basis der OSGi Service Plattform zu entwerfen und einzusetzen. Durch dieses Konzept könnte nicht nur die Funktionalität der Verwaltungskomponente zur Laufzeit verändert werden. Weiterhin ist es denkbar, dass diese erweiterte Funktionalität auch in der Erfassungskomponente zur Verfügung steht. Ferner kann untersucht werden, welche Konsequenzen eine solche Infrastruktur auf die Ausspielungskomponente hat. Insbesondere in Hinblick auf Apache Sling, da dieses auch auf der OSGi Services Plattform basiert und sich dadurch Synergien ergeben könnten, wobei in der Praxis die Ausspielungskomponente oft als verteiltes System realisiert ist. Dies ergibt sich aus bereits aus der Lastverteilung. Aus diesem Grund müssten die OSGi Services auch verteilt arbeiten.

Anhang A

Der Anhang beinhaltet eine Darstellung des Inhalts der beigefügten CD sowie das Abkürzungs- , das Abbildungs- und das Literaturverzeichnis.

A.1 Inhalt der beigelegten CD

Auf der beigelegten CD befindet sich zunächst in dem Ordner `Bachelorarbeit` diese Arbeit in dem PDF Format.
Darüber hinaus befinden sich in der Datei `Sourcen.zip` die Java-Projekte, die im Rahmen dieser Arbeit erstellt worden sind. Dieses Archiv ist direkt in einen Eclipse Workspace importierbar. Dies geschieht über die Import Funktion von Eclipse. In dieser muss die Option „Existing Projects into Workspace“ ausgewählt werden. Um die Projekte innerhalb von Eclipse verwenden zu können muss das Eclipse Plugin „Maven for Eclipse integration“ installiert sein. Anschließend müssen einige absolute Pfadangaben angepasst werden. Insbesondere in der Konfigurationsdatei des Projekts Felix. Wird Felix direkt aus Eclipse heraus gestartet, muss in der Run Configuration hierzu das Vm Argument *-Dfelix.config.properties=file:conf/config.properties* hinzugefügt werden.

In dem Verzeichnis `EventServer` befinden sich die ausführbaren Dateien. Dieses muss vor der Ausführung auf die lokale Festplatte kopiert werden. Eine Ausführung von der CD ist nicht möglich. In der Datei `readme.txt` stehen weitere Informationen zum Starten der Anwendung.
Durch die Datei `start.bat` in dem Ordner `EventServer` wird der explorativen Prototyp und der EventServer gestartet. Darüber hinaus öffnet sich eine grafische Anwendung mit dessen Hilfe die jUnit4Osgi Testfälle ausgeführt werden können.

A.2 Abkürzungen

API	Application Programming Interface
BLOB	Binary large Object
CMS	Content Management System
CND	Content Type Definition
CRX	Content Repository Extreme
DI	Dependency Injection
IDE	Integrated Development Environment
IoC	Inversion Of Control
JAR	Java Archiv
JCP	Java Community Process
JCR	Java Content Repository
JRE	Java Runtime Environment
JSR	Java Specification Request
JTA	Java Transaction API
JVM	Java Virtual Machine
POJO	Plain Old Java Object
REST	Representational State Transfer
RMI	Remote Methode Invocation
SCR	Service Component Runtime
Spring DM	Spring Dynamic Modules
SOA	Service Oriented Architecture
SOC	Service Oriented Computing
TUI	Textual User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WMC	Apache Felix Web Management Console

Abbildungsverzeichnis

Literaturverzeichnis

[Bar09a] BARTLETT, Neil: *OSGi in Practice*. Bd. January 11, 2009. Abgerufen am 09.10.2009. 2009 http://neilbartlett.name/blog/osgibook/

[Bar09b] BARTLETT, Neil: *Reasons Not to Mourn JSR 294*. Online. http://neilbartlett.name/blog/2009/12/11/reasons-not-to-mourn-jsr-294/. Abgerufen am 22.12.2009, 2009

[Ble09] BLEWITT, Alex: *JSR294 is dead, long live OSGi!* Online. http://alblue.blogspot.com/2009/12/jsr294-is-dead-long-live-osgi.html. Abgerufen am 22.12.2009, 2009

[Boi05] BOIKO, Bob ; WEBB, Chris (Hrsg.): *Content Management Bible, 2nd Edition*. Wiley Publishing, Inc., 2005. – ISBN 0–7645–7371–3

[Buc09] BUCKLEY, Alex: *JSR 294 Inactive*. http://altair.cs.oswego.edu/pipermail/jsr294-modularity-eg/2009-December/000392.html. Abgerufen am 22.12.2009, 12 2009

[DJC08] DAN, Asit ; JOHNSON, Robert D. ; CARRATO, Tony: SOA service reuse by design. In: *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–029–6, S. 25–28

[EHL07] ESCOFFIER, C. ; HALL, R. S. ; LALANDA, P.: iPOJO: an Extensible Service-Oriented Component Framework. In: *Proc. IEEE International Conference on Services Computing SCC 2007*, 2007, S. 474–481

[Erl09] ERL, Thomas ; TAUB, Mark L. (Hrsg.): *SOA Design Patterns*. PRENTICE HALL, 2009. – ISBN 978–0–13–613516–6

[Fer07] FERGUSON, Arron: *Creating Content Management Systems in Java.* Charles River Media, 2007. – ISBN 1–58450–466–8

[Fie00] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*. Abgerufen am 04.12.2009. Dissertation, 2000

[Fow04] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. Online. http://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection. Abgerufen am 09.10.2009, 2004

[Gen08] GENTZ, Ekkehard: *Logging In OSGi Enterprise Anwendungen*. Online. http://web.mac.com/ekkehard.gentz/ekkes-corner/blog/Eintr%C3%A4ge/2008/9/24_Logging_in_OSGI_Enterprise_Anwendungen%2C_Teil_1.html. Abgerufen am 04.02.2010, 2008

[GHJV04] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuser*. Addison-Wesley, 2004. – ISBN 0–201–63361–2

[GP70] GAUTHIER, Richard ; PONTO, Stephan: *Designing systems programs*. Prentice Hall; First Edition edition (November 27, 1970), 1970. – 274 S. – ISBN 978–0132019620

[iPOa] *Apache Felix iPOJO Online Manipulator*. Online. http://felix.apache.org/site/apache-felix-ipojo-online-manipulator.html. Abgerufen am 10.10.2009

[iPOb] *How to write your iPOJO Handler*. Online. http://felix.apache.org/site/how-to-write-your-own-handler.html. Abgerufen am 10.10.2009

[JDK09] *JDK 7 Features*. Online. http://openjdk.java.net/projects/jdk7/features/. Abgerufen am 22.12.2009, 2009

[JSR] *Java Specification Request 170*. Online. http://jcp.org/en/jsr/detail?id=170. Abgerufen am 10.09.2009

[Kri08] KRIENS, Peter: How OSGi Changed My Life. In: *Queue* 6 (2008), Nr. 1, S. 44–51. http://dx.doi.org/http://doi.acm.org/10.1145/1348583.1348594. – DOI http://doi.acm.org/10.1145/1348583.1348594. – ISSN 1542–7730

[Kum02] KUMFERT, T G ;. G ; Epperly E. G ; Epperly: Software in the DOE: The Hidden Overhead of"The Build" / Lawrence Livermore National Lab., CA (US). Abgerufen am 29.11.2009, 2002. https://e-reports-ext.llnl.gov/pdf/244668.pdf. 2002. – Forschungsbericht

[Mah05] MAHMOUD, Qusay H.: Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). In: *Sun Developer Network* 1 (2005). http://java.sun.com/developer/technicalArticles/WebServices/soa/

[Mil09] MILLER, Alex: *Java SE 7 Preview*. http://puredanger.com/techfiles/090204/Java7Preview.pdf. Abgerufen am 22.12.2009, 2009

[NP09] NUESCHELER, David. ; PIEGAZE, Peeter ; DAY SOFTWARE (Hrsg.): *jsr170-1.0 Specification*. Abgerufen am 09.10.2009. Day Software, 05 2009. http://jcp.org/aboutJava/communityprocess/final/jsr170/index.html

[Orn04] ORNA, Elizabeth: *How to Develop an Organizational Information Strategy*. Gower Publishing Co Ltd, 2004. – ISBN 978–0566085796

[OSG07] OSGI SERVICE ALLIANCE: OSGi Technical WhiePaper. Version: 2007. http://www.osgi.org/wiki/uploads/CommunityEvent2007/OSGiBestPractices.pdf. 2007. – Forschungsbericht

[OSG09a] *OSGi Markets and Solutions*. Online. http://www.osgi.org/Markets/HomePage. Abgerufen am 09.10.2009, 2009

[OSG09b] THE OSGI ALLIANCE: OSGi Service Platform Core Specification. Abgerufen am 09.10.2009, 2009. http://www.osgi.org/download/r4v42/r4.core.pdf. 2009. – Forschungsbericht

[OSG09c] OSGI ALLIANCE: OSGi Service Platform Service Compendium. Abgerufen am 04.12.2009, 2009. http://www.osgi.org/download/r4v42/r4.cmpn.pdf. 2009. – Forschungsbericht

[Rei08] REINHOLD, Mark: *There is not a moment to lose!* Online. http://blogs.sun.com/mr/entry/jigsaw. Abgerufen am 22.12.2009, 12 2008

[RR02] ROTHFUSS, Gunther ; RIED, Christian: *Content Management mit XML*. Bd. Auflage: 2., überarb. A. (27. November 2002). Springer, Berlin, 2002. – ISBN 3–540–43844–0

[Rub09] RUBIO, Daniel: *Pro Spring Dynamic Modules for OSGi Service Platforms*. Berkely, CA, USA : Apress, 2009. – ISBN 1430216123, 9781430216124

[SLF] *SLF4J Manual*. http://www.slf4j.org/manual.html

[SM] SUN MICROSYSTEMS, Inc.: *How Classes are Found*. Online. `http://java.sun.com/javase/6/docs/technotes/tools/findingclasses.html`. Abgerufen am 12.10.2009

[SMP98] SOCIETY OF MOTION PICTURE AND TELEVISION ENGINEERS, EUROPEAN BROADCASTING UNION: Task Force for Harmonized Standards for the Exchange of Program Material as Bit Streams. Abgerufen am 09.10.2009, 1998. `http://www.smpte.org/standards/tf_home/Final_Report_-_Findings.pdf`. 1998. – Forschungsbericht

[Wal09] WALLS, Craig: *Modular Java. Creating Flexible Application with OSGi and Spring*. The Üragmatice Programmers, 2009. – ISBN 978–1934356–40–1

[WHKL08] WÜTHERICH, Gerd ; HARTMANN, Nils ; KOLB, Bernd ; LÜBKEN, Matthias: *Die OSGi Service Plattfrom*. dpunkt.verlag GmbH, 2008. – ISBN 978–3–89864–457

[Zhu05] ZHU, Haibin: Challenges to reusable services. In: *Proc. IEEE International Conference on Services Computing* Bd. 2, 2005, S. 243–244

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 09.02.2010 Steffen Kuche