INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Java EE Applikationsmigration mittels einer Ontologie

Dominik Billing

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Java EE Applikationsmigration mittels einer Ontologie

Dominik Billing

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde
Abgabetermin:	24. November 2015

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24. November 2015

.....

(Dominik Billing)

Danksagung

Ich möchte mich an dieser Stelle bei all denen bedanken, die mich bei der Anfertigung meiner Masterarbeit und während des kompletten Studiums unterstützt haben.
Hierbei ist vor allem mein Betreuer Nils gentschen Felder zu nennen, der mir sowohl bei inhaltlichen Fragen als auch konzeptionell immer zur Seite stand.
Meinen Kollegen bei der PROSTEP AG sei hier auch ein großes Dankeschön ausgesprochen für die Hilfe bei der Themenfindung und Ermöglichung des Studiums parallel zur Arbeit, was nicht selbstverständlich ist.
Ein großes Danke gilt auch meinen Eltern für ihre Unterstützung und kurzfristigen Korrekturen an dieser Arbeit.
Mein ganz besonderer Dank gilt meiner Freundin Sonja Fischer, die meine dauernde geistige Abwesenheit ertragen hat und dennoch bereit war dieses umfangreiche Manuskript Korrektur zu lesen.
Bedanken möchte ich mich auch besonders bei Thomas Fischer, ohne den mein Studium zweifellos niemals in dieser Form möglich gewesen wäre.

Abstract

Die Migration von Java EE Applikationen von einem auf einen anderen Anwendungsserver stellt aufgrund der großen Unterschiede zwischen verschiedenen Anwendungsservern eine Herausforderung dar. Zusätzlich sind die meisten Java EE Applikationen durch Verbesserungen oder Vereinfachungen nicht komplett Java EE kompatibel und damit für einen spezifischen Anwendungsserver ausgelegt.
Diese Arbeit hat das Ziel, Probleme bei der Migration von Java EE Applikationen einfach, automatisiert und für jeden beliebigen Server aufzuzeigen. Hierfür wird eine Ontologie entwickelt und an theoretischen Daten evaluiert. Bei der Entwicklung dieser Ontologie wird sich an den Methoden von Uschold und King, Grüninger und Fox und METHONTOLOGY orientiert. Schließlich wird ein Prototyp entwickelt, der diese Ontologie als Grundlage verwendet.
Die Implementierung der Ontologie wird anhand der formalen Ontologiesprache OWL durchgeführt. Dabei kann das Problem der Open-World Annahme von Ontologien durch weniger komplexe und angepasste Abfragen umgangen werden. Abschließend wird die Ontologie an einem komplexen Standardbeispiel von Java EE 6 erfolgreich getestet. Die Resultate zeigen die erwarteten Inkompatibilitäten zwischen dem Standardbeispiel und unterschiedlichen Servern auf.

Inhaltsverzeichnis

1 Motivation

Der Java EE Standard (Java Platform, Enterprise Edition, auch J2EE) stellt eine Spezifikation für in Java programmierte Anwendungen dar [Ora13d]. Dieser Standard besteht aus unterschiedlichen Java EE APIs (application programming interfaces - Programmierschnittstellen). Jede dieser APIs definiert die Schnittstelle zu einem festgelegten Funktionsumfang von Java EE wie beispielsweise JavaMail als Mail Schnittstelle oder Java Persistence API für die Datenbankverbindung. Jeder Java EE Applikationsserver implementiert einen Teil dieser APIs, was zu sehr vielen verschiedenen Implementierungen für jede API führt. Auf diese Weise bestehen zwischen Java EE Anwendungsservern teilweise starke Unterschiede in der Leistung und der Konfiguration. Nicht alle Anwendungsserver decken alle unterschiedlichen Aspekte von Applikationen ab oder lassen Teile des Java EE Standards aus. Das führt neben unterschiedlicher Implementierungen der Applikationsserver zu Unterschieden im Deployment von Anwendungen auf diesen Servern. In diesem Zusammenhang liefern manche Firmen ihre Produkte mit Servern aus, die angepasste Konfigurationen oder geschützte Funktionalitäten mitbringen, die auf anderen Servern nicht oder nur teilweise vorhanden sind. [Orag]

Üblicherweise wird eine Applikation für einen bestimmten Applikationsserver entwickelt, weil dieser Vorteile gegenüber anderen Servern in einem für die Applikation wichtigen Bereich aufweist. Aus diesem Grund können die wenigsten Applikationen, obwohl sie nach einem übergreifenden Standard entwickelt wurden, nicht von einem Anwendungsserver auf einen anderen übertragen werden. In diesem Fall sind Migrationen mit teilweise sehr hohem Aufwand nötig. Hierbei müssen beispielsweise Konventionen für Dateinamen, Message-Syntax, Konfigurationen, Java EE APIs oder Ähnliches betrachtet werden. [Orag]

Solche Migrationen sind dann notwendig, wenn Applikationen erweitert werden müssen und der aktuell verwendete Applikationsserver die Voraussetzungen oder benötigten APIs nicht unterstützt. Andere Gründe können beispielsweise Sicherheitsaspekte oder nicht mehr weiter entwickelte veraltete Server sein, deren Support nicht länger gewährleistet ist. Im kommerziellen Umfeld sind finanzielle Überlegungen meist der Grund für eine Migration, wenn der Unterhalt eines Servers auf lange Sicht teurer ist als eine Migration auf einen billigeren Anwendungsserver. [Sag]

Bei einer Migration wird im Allgemeinen damit begonnen, dass die Applikation mit ihren Konfigurationen, Source-Dateien, etc. analysiert wird, mit dem Ziel herauszufinden, ob eine Migration grundsätzlich möglich ist. Wenn zu dem Schluss gekommen wurde, dass eine Migration möglich ist, werden anschließend die Konfiguration, die Source-Dateien und die Struktur der Anwendung so geändert, dass sie auf dem Zielserver installierbar ist. Anschließend muss die Funktionalität der Applikation intensiv getestet werden. Am Ende der Migration steht das fertige Paket, das auf dem Zielserver installiert werden kann.

Diese Schritte können beliebig kompliziert sein und setzen ein umfangreiches Wissen über die Applikation sowie den Quell- als auch den Zielserver voraus. Da dies im Normalfall nicht gegeben ist beziehungsweise man sich dieses Wissen erst angeeignet muss, stellen Migrationen

eine sehr komplexe Aufgabe dar, die nur mit sehr hohem Aufwand erledigt werden können.
Im Internet existiert eine Vielzahl von Artikeln zur Migration einer Anwendung von einem speziellen Applikationsserver auf einen anderen, wie beispielsweise „How To Migrate Your Weblogic or WebSphere App to Tomcat“ [Bri]. Ebenso gibt es von Herstellern einiger Server auch komplette Bücher oder Toolboxen zur Migration auf ihren eigenen Server. Als Beispiele zählen hierfür etwa „WebSphere Application - Server V8.5 Migration Guide“ [ACE+12] oder das „WebSphere Application Server Migration Toolkit“[IBMb]. Speziell die Hersteller von eigenen Anwendungsservern wie Oracle mit Weblogic [Rom] und GlassFish [Orae], IBM mit WebSphere [ACE+12] oder Red Hat mit JBoss [Redb] stellen Informationen und Migrationstools zur Verfügung, die beschreiben wie man von konkurrierenden Produkten auf ihre Produkte migrieren kann. Entwickler sind also darauf angewiesen, dass die dort gefundenen Informationen alle anderen Server und Fälle abdecken. Es existiert kein ganzheitliches Tool, das alle Quell- und Zielserver abdeckt.
Aufgrund dieses hohen Migrationsaufwandes und der sehr selten ausreichenden Informationen bezüglich der Anwendung selbst und den entsprechenden Applikationsservern soll im Rahmen dieser Arbeit ein Hilfsmittel zur einfachen Analyse der Anwendung entwickelt werden. Insbesondere soll hier eine automatische Klassifizierung und Analyse von Applikationen anhand von Eigenschaften der Anwendungen vorgenommen werden. Die gefundenen Informationen der Applikation werden strukturiert in einer Ontologie im OWL Format dargestellt. Auf diese Weise wird die Anwendung detailliert analysiert und ihre für eine Migration relevanten Eigenschaften objektiv in der Ontologie abgebildet.
Für diesen Ansatz wurde eine Ontologie aufgrund der Möglichkeit zur Migration von Daten und als potentielle Metasprache gewählt [SVS12; UG96]. Ontologien haben gegenüber einfachen Taxonomien den Vorteil, nicht nur Subklassen-Relationen zu besitzen. Es sind alle Arten von Relationen möglich. Zusätzlich können über einfache Axiome und Regeln zusätzliche Zusammenhänge direkt aufgezeigt werden. Das verringert höheren Implementierungsaufwand, um selbst Schlussfolgerungen zu ziehen.
Die hierfür verwendete Ontologie soll in dieser Arbeit entwickelt werden. Durch das Abgleichen der gefundenen Informationen der analysierten Applikation mit entsprechenden Informationen von Anwendungsservern sollen mögliche Probleme aufgedeckt und beschrieben werden. Auf diese Weise kann der Migrationsaufwand besser abgeschätzt werden und eine eventuelle absolute Inkompatibilität erkannt werden.
Die vorliegende Arbeit gliedert sich grob in die Schritte der von Uschold und King in [UG96] vorgestellte Methode zur Entwicklung einer Ontologie und die tatsächliche Implementierung eines Prototypen. In dieser Motivation wurde der Zweck und die Verwendung einer Ontologie im Rahmen der Java EE Applikationsmigration aufgezeigt. Im folgenden Kapitel werden grundlegende Eigenschaften von Java EE Applikationen, Java EE Anwendungsserver und Ontologien beschrieben (siehe Kapitel 2). Anschließend wird in Kapitel 3 auf ähnliche Arbeiten im Bereich der Daten- und Anwendungsmigration sowie der Ontologien eingegangen. Kapitel 4 beschäftigt sich mit den unterschiedlichen Methoden zur Entwicklung einer Ontologie. Hier wird zusätzlich aufgezeigt, warum eine Ontologie in diesem Zusammenhang sinnvoll ist, welche Bestandteile sie besitzen soll und mit welcher der vorgestellten Methoden gearbeitet wird. In Kapitel 5 wird das konzeptionelle Modell der Ontologie entwickelt, wozu neben den tatsächlichen Elementen wie Klassen, Relationen und Regeln auch Kompetenzfragen zur Evaluation der Ontologie gehören. Die tatsächliche Implementierung der vorher entwickelten Ontologie erfolgt in Kapitel 6. Zusätzlich wird die entwickelte Ontologie dort anhand der Kompetenzfragen evaluiert. Kapitel 7 beschäftigt sich mit der Implementierung

eines Prototypen, der Java EE Applikationen analysiert und die gefundenen Eigenschaften in der entwickelten Ontologie speichert. In diesem Kapitel wird die Software anhand eines einfachen Beispiels angewendet und deren Ergebnisse überprüft. Anschließend werden die Resultate dieser Arbeit in Kapitel 8 zusammengefasst und ein Ausblick über zukünftige Entwicklungen gegeben.

2 Grundlagen

Für die Migration von *Java Platform, Enterprise Edition* (Java EE) Anwendungen ist es nötig die vorhandene Anwendung genau zu analysieren. In der folgenden Auflistung finden sich die in dieser Arbeit verwendeten Begriffe wieder:

Java Platform, Enterprise Edition

Unter *Java Platform, Enterprise Edition* versteht man den Standard [DS13b], mit dem Geschäftslogiken in Java abgebildet werden sollen. Abgekürzt wird der Standard entweder mit *Java EE* oder *J2EE*, wobei J2EE bis zur Version 1.4 und Java EE ab der danach folgenden Version 5 gebräuchlich ist. In diesem Standard werden unterschiedliche Schnittstellen (*APIs*) definiert, die es einem Entwickler einfacher machen sollen unterschiedliche Anwendungsgebiete zu erfüllen. Der Java EE Standard ist in zwei unterschiedliche sogenannte Profile gegliedert: Das *Full Profile*, in dem alle definierten APIs enthalten sind, und das *Web Profile*, das nur die für Webanwendungen relevanten APIs beinhaltet. Eine detaillierte Beschreibung befindet sich in Kapitel 2.1.

Java EE API

Unter einer *API* versteht man eine Programmierschnittstelle oder englisch *application programming interface*. Java EE APIs sind die im Java EE Standard definierten APIs. Eine Übersicht über die in Java EE 7 definierten APIs liefert Tabelle 2.1.

Java EE Anwendung

Eine Java EE Anwendung wird auf einem Java EE Anwendungsserver ausgeführt. Die Applikation selbst besteht aus einem Container, der neben den verwendeten Klassen und Programmdateien auch Konfigurationsdateien und zusätzliche Informationen enthält. Diese Applikationspakete werden genauer in Abschnitt 2.2.2 beschrieben.

Java EE Anwendungsserver

Unter einem Java EE Anwendungsserver versteht man ein Ausführungsplattform für Java EE Anwendungen. Das bedeutet auch, dass jeder solche Server entweder das Full oder das Web Profile implementieren muss. Auf eine Auswahl an Java EE Applikationsserver wird in Kapitel 2.3 eingegangen.

Deployment Deskriptoren

Deployment Deskriptoren sind Konfigurationsdateien von Java EE Anwendungen, die dem Anwendungsserver Informationen über das Deployen, also das Installieren, der Anwendung mitteilen. Darin können beispielsweise Datenbankverbindungen oder der Name der Anwendung definiert werden. Auf die im Java EE Standard definierten Deployment Deskriptoren wird in Abschnitt 2.2.3 eingegangen. Zusätzlich gibt es für die

meisten Applikationsserver noch individuelle Deployment Deskriptoren, mit denen sich befasst wird, wenn die Server kurz beschrieben werden (siehe Kapitel 2.3).

Servlet

Ein *Servlet* ist eine Webkomponente, die in Java programmiert wurde. Es wird von einem Applikationsserver gesteuert, der dynamische Inhalte erzeugt und im Hintergrund die Geschäftslogik auf dem Server berechnet. [CM13]

Applet

Ein *Applet* ist eine Benutzerschnittstellenkomponente, die in Java programmiert wurde. Es wird im Webbrowser des Anwenders ausgeführt und berechnet die Geschäftslogik auf dem Rechner des Anwenders. [DS13b]

Nach der Einführung in Java EE wird in Kapitel 2.4 der Begriff der Ontologie definiert und Möglichkeiten zur Repräsentation einer Ontologie aufgezeigt.

2.1 Java Platform, Enterprise Edition

In diesem Kapitel wird der *Java EE* Standard genauer beleuchtet und auf die darin enthaltenen Schnittstellen kurz eingegangen. Die folgende Tabelle 2.1 zeigt die im Full und Web Profile definierten Programmierschnittstellen, hier mit *API* für *application programming interface* abgekürzt [Ora14a]. Die im Web Profile vorhandenen APIs sind mit $*$ markiert.

Tabelle 2.1: Übersicht über die Entwicklung der verwendeten APIs im Java EE Standard (Full Profile) von Java EE 5 bis Java EE 7 [DS13b; DS13a; CS09; Sha06]. $*$ markiert die im Web Profile von Java EE 7 enthaltenen APIs.

APIs		**API Version**		
Abkürzung		**Java EE 7**	**Java EE 6**	**Java EE 5**
EJB $*$	Enterprise JavaBeans	3.2	3.1	3.0
Servlet $*$	Java Servlet	3.1	3.0	2.5
JSP $*$	JavaServer Pages	2.3	2.2	2.1
EL $*$	Expression Language	3.0	2.2	
JMS	Java Message Service API	2.0	1.1	1.1
JTA $*$	Java Transaction API	1.2	1.1	1.1
JavaMail	JavaMail API	1.5	1.4	1.4
JAF	JavaBeans Activition Framework			1.1
Connector	Java EE Connector Architecture	1.7	1.6	1.5
Web Services		1.4	1.3	1.2
JAX-RPC	Java API for XML-based RPC	1.1	1.1	1.1
JAX-WS	Java API for XML-Based Web Services	2.2	2.2	2.0
JAX-RS $*$	Java API for RESTful Web Services	2.0	1.1	
WebSocket $*$	Java API for WebSocket	1.0		
JSON-P $*$	Java API for JSON Processing	1.0		

Tabelle 2.1: Übersicht über die Entwicklung der verwendeten APIs im Java EE Standard (Full Profile) von Java EE 5 bis Java EE 7 [DS13b; DS13a; CS09; Sha06].
∗ markiert die im Web Profile von Java EE 7 enthaltenen APIs. – Fortsetzung

APIs		**API Version**		
Abkürzung		**Java EE 7**	**Java EE 6**	**Java EE 5**
Concurrency Utilities	Concurrency Utilities for Java EE	1.0		
Batch	Batch Applications for the Java Platform	1.0		
JAXB	Java Architecture for XML Binding	2.2	2.2	2.0
JAXR	Java API for XML Registries			1.0
SAAJ	SOAP with Attachments API for Java			1.3
Java EE Management		1.1	1.1	1.1
JACC	Java Authorization Service Provider Contract for Containers	1.5	1.4	1.1
JASPIC	Java Authentication Service Provider Interface for Containers	1.1	1.0	
JSTL ∗	JavaServer Pages Standard Tag Library	1.2	1.2	1.2
JSF ∗	JavaServer Faces	2.2	2.0	1.2
Common Annotations ∗	Common Annotations for the Java Platform	1.2	1.1	1.0
JPA ∗	Java Persistence API	2.1	2.0	1.0
Bean Validation ∗		1.1	1.0	
Managed Beans ∗		1.0	1.0	
Interceptors ∗		1.2	1.1	
Context and Dependency Injection for Java EE ∗		1.1	1.0	
Dependency Injection for Java ∗		1.0	1.0	

Tabelle 2.1 bildet einen Überblick über die in den Versionen 5 bis 7 in Java EE vorhandenen APIs und deren Versionen. Für jede dieser APIs gibt es eine technische Spezifikation (Java Specification Request - JSR) ebenso wie für Java EE selbst. Java EE 7 ist beispielsweise in JSR-342 [DS13b] definiert oder EJB 3.2 in JSR-345 [CM13]. Java EE wird vom *Java Community Process* (JCP) weiterentwickelt. Dem JCP gehören beispielsweise AOL, AT&T, Deutsche Telekom, Eclipse, Ericsson, Fraunhofer Institut, Fujitsu, Google, HP, IBM, Intel, LG, Mitsubishi, Motorola, Nokia, Novell, Oracle, Red Hat, Samsung, SAP, Sony, VMWare oder Vodafone an. Die einzelnen JSRs werden von unterschiedlichen Unternehmen oder Entwicklergruppen definiert und entwickelt. Durch gegenseitige Kontrolle und gemischte Projektteams wird innerhalb des JCP sichergestellt, dass die Technologien weiterhin stabil und plattformübergreifend funktionsfähig bleiben und entsprechend weiterentwickelt werden. [Ora14a; Sal14]
Die in Tabelle 2.1 gezeigten Java EE APIs können nach [DS13b] in die folgenden unterschiedlichen Bereiche eingeteilt werden:

HTTP/HTTPS

Servlet, *JavaServer Pages* (JSP), *JavaServer Faces* (JSF) und Webservice APIs bilden die HTTP und HTTPS Schnittstelle des Java EE Standards. Die verwendeten Klassen sind alle im Paket `java.net` hinterlegt.

Java Transaction API (JTA)

Die *Java Transaction API* ist eine Schnittstelle, die auf Anwendungsseite Transaktionen begrenzt und zwischen Transaktionsmanager und Ressourcenmanager von den *Java EE Service Provider Interfaces* (SPIs) des Applikationsservers genutzt wird. Die SPIs stellen eine Schnittstelle zwischen der Java EE Plattform und externen Diensten zur Verfügung, die auf die Applikation zugreifen wollen.

Java Persistence API (JPA)

Die *Java Persistence API* ist die Standardschnittstelle, um Persistenz und Objektbeziehungsabbildungen zu verwalten.

Java Message Service (JMS)

Der *Java Message Service* ist als Standardschnittstelle für das Nachrichtensystem verantwortlich und unterstützt verlässliche Punkt-zu-Punkt Nachrichten sowie das „Publish-Subscribe“ Model.

JavaMail

JavaMail ist die Schnittstelle, mit der E-Mails verschickt werden können.

Java EE Connector Architecture (JCA)

Die *Java EE Connector Architecture* erlaubt es Ressourcenadapter, die auf Informationssysteme zugreifen, in eine Java EE Anwendung zu integrieren.

Sicherheitsdienste

Zu den Sicherheitsschnittstellen gehören die Java EE APIs *Java Authorization Service Provider Contract for Containers* (JACC) und *Java Authentication Service Provider Interface for Containers* (JASPIC).

Webservices

Im Bereich der Webservices zählen die APIs *Web Services*, *Java API for XML Web Services* (JAX-WS) und *Java API for XML-based RPC* (JAX-RPC) zu den Schnittstellen, die Webservices über HTTP oder SOAP unterstützen. Mit *Java Architecture for XML Binding* (JAXB) wird eine Abbildung zwischen Java Klassen und dem bei SOAP verwendetem XML unterstützt. Mit *SOAP with Attachments API for Java* (SAAJ) können SOAP Nachrichten verwaltet werden. Durch *Java API for XML Registries* (JAXR) wird Zugriff auf XML Registerserver realisiert. Die *Java API for JSON Processing* (JSON-P) stellt eine Möglichkeit zur Verfügung, um JSON Text zu verarbeiten. *Java API for WebSocket* (WebSocket) ist die Schnittstelle, um WebSocket Anwendungen zu erstellen. Mit *Java API for RESTful Web Services* (JAX-RS) wird Unterstützung für Webservices, die über REST kommunizieren, zur Verfügung gestellt.

Concurrency Utilities

Mit den *Concurrency Utilities for Java EE* ist eine Schnittstelle gegeben, die Threads verwalten kann.

Batch

Die Schnittstelle *Batch Applications for the Java Plaform* (Batch) ermöglicht die Erstellung von Hintergrundanwendungen und eine Schnittstelle, um Aufgaben zu planen und auszuführen.

Verwaltung

Die Schnittstellen *Java Management Extensions* (JMX) und *Java EE Management* stellen Unterstützung für das Verwalten von Software zur Verfügung.

Wenn ein Java EE Server alle Container und Dienste der Java EE Spezifikation fehlerfrei anbietet, kann er als vollständig Java EE konformer Server zertifiziert werden. Es gibt aber auch Java EE Server, die nicht die vollständige Java EE Spezifikation implementieren, sondern nur einen Teil davon. Neben dem Full Profile für den kompletten Java EE Standard gibt es noch das Web Profile, das die Untermenge an Java EE Technologien beinhaltet, die für die Implementierung der gebräuchlichsten Java EE Anwendungen erforderlich sind. In Tabelle 2.1 sind alle APIs aufgezählt, die im Full Profile benötigt sind. Die mit $*$ markierten APIs werden im Web Profile definiert. In Kapitel 2.3 wird näher auf eine kleine Auswahl an unterschiedlichen Java Applikationsserver eingegangen, die den Java EE Standard im Full oder Web Profile unterstützen. [DS13b; DS13a; Sal14]

2.2 Java EE Anwendungen

Java EE Anwendungen sind Container von Programmen und Konfigurationen, die auf einem Java EE Anwendungsserver ausgeführt werden können. Diese Anwendungen können im Normalfall nicht einfach so gestartet werden, da sie auf die Implementierungen der Java EE APIs aus dem Applikationsserver zurückgreifen.
In diesem Kapitel wird auf die unterschiedlichen Arten von Anwendungen und deren Struktur eingegangen. Anschließend werden die für den Anwendungsserver wichtigen Konfigurationsdateien, die Deployment Deskriptoren, beschrieben, die im Java EE Standard definiert sind.

2.2.1 Java EE Anwendungsarten

Auf einem Java EE Anwendungsserver können unterschiedliche Arten von Anwendungen installiert werden. Die folgende Aufzählung gibt einen kurzen Überblick über diese Typen von Anwendungen und Modulen: [DS13b; Vat13]

- **Enterprise Anwendung**

  Unter einer *Enterprise Anwendung* versteht man eine Java EE Anwendung, die das Full Profile des Java EE Standards benötigt, um lauffähig zu sein. In vielen Fällen sind einzelne oder mehrere der nachfolgenden Anwendungstypen Bestandteile einer Enterprise Anwendung. Eine Enterprise Anwendung besteht aus einem Anwendungspaket mit der

Dateiendung `ear` (siehe Kapitel 2.2.2.4). Der Java EE konforme Deployment Deskriptor für eine Enterprise Anwendung heißt `application.xml` und wird in Kapitel 2.2.3.2 beschrieben.

- **Web Modul**

  Eine *Webanwendung* ist eine Java EE Anwendung, die nicht das Full Profile, aber zumindest das Web Profile des Java EE Standards voraussetzt. Webanwendungen sind Komponenten, die in den meisten Fällen Hintergrundlogik eines Geschäftsprozesses bereitstellen und mit dem dynamischen Frontend zusammenarbeiten. Die Hauptkomponente von Webanwendungen sind Servlets, weshalb die entsprechende Spezifikation von Webanwendungen gleichzeitig auch die Spezifikation von Java Servlets ist [CM13]. Das Applikationspaket einer Webanwendung hat die Dateiendung `war` (siehe Kapitel 2.2.2.2) und der zugehörige Deployment Deskriptor heißt `web.xml` (siehe Kapitel 2.2.3.1).

- **Anwendungsclient**

  Anwendungsclient Module sind Java Programme, die üblicherweise als Benutzeroberfläche auf dem Computer des Anwenders ausgeführt werden. Im Gegensatz zu Webanwendungen, bieten Anwendungsclient Module einen ähnlichen Komfort wie normale Desktopapplikationen und können dennoch auf alle Java EE Logiken zugreifen. Anwendungsclient Module sind normale `jar`-Dateien (siehe Kapitel 2.2.2.1) und besitzen den eigenen Deployment Deskriptor `application-client.xml`.

- **Ressourcen Adapter Modul**

  Ein *Ressourcen Adapter* ist eine Softwarekomponente, die auf Systemebene beispielsweise Netzwerkverbindungen zu externen Ressourcen verwaltet und implementiert. Über einen Ressourcen Adapter können andere Java EE Anwendungen beispielsweise in einer Enterprise Anwendung auf die gewünschten Ressourcen zugreifen. Ein Ressourcen Adapter Modul ist ein Container mit der Dateiendung `rar` (siehe Kapitel 2.2.2.3) und hat den eigenen Deployment Deskriptor `rar.xml`.

- **Enterprise JavaBeans**

  Enterprise JavaBeans (EJBs) sind Komponenten, die objektorientierte Geschäftsanwendungen in der Programmiersprache Java darstellen. EJBs unterstützen Webservices und bieten über standardisierte Schnittstellen die Möglichkeit mit anderen Java EE und nicht-Java Anwendungen zu interagieren. Viele Webanwendungen bestehen auch aus EJBs. Ein EJB ist ein `jar`- oder `war`-Paket mit dem Deployment Deskriptor `ejb-jar.xml`.

2.2.2 Anwendungspakete

In diesem Kapitel wird die generelle Struktur von Java EE Applikationen beschrieben. Hierbei ist zwischen den Typen *jar* (Java ARchive), *war* (Web application ARchive), *rar* (Ressource ARchive) und *ear* (Enterprise ARchive) zu unterscheiden. Der Grundtyp eines Java Programms bzw. einer Java Bibliothek bildet das `jar`-Archiv. Jeder andere Typ ist strukturell vom `jar`-Paket abgeleitet und damit ein `zip`-Archiv mit spezieller Struktur. Ein `war`-Archiv enthält eine Webapplikation, die wiederum Java-Klassen, Java-Servlets, JavaServer Pages, XML-Dateien und statischen Inhalt beinhaltet. In einem `rar`-Paket befinden sich Ressourcen

Adapter, die eine Schnittstelle zu externen Ressourcen bereitstellen. Enterprise-Archive enthalten mehrere Module, worunter auch Webapplikationen oder Ressourcenanwendungen fallen können. [Sal14]

2.2.2.1 jar-Archiv

Das `jar`-Paket ist das Standardformat für Java Bibliotheken. Ein `jar`-Archiv ist ein `zip`-Archiv, in dem kompilierte Java-Klassen als `class`-Dateien enthalten sind. Im optionalen Ordner `META-INF` können Paket- und Erweiterungskonfigurationen enthalten sein. Die wohl bekannteste Konfigurationsdatei `META-INF/MANIFEST.MF` enthält Informationen zum Paket selbst beispielsweise wie die verwendete Java Version oder benötigte andere Bibliotheken. Prinzipiell können Java Programme auch direkt über die `class`-Dateien gestartet werden, es ist aber bei größeren Programmen sehr viel einfacher alles in ein oder mehrere `jar`-Pakete zu packen. [KS08; DS13b]

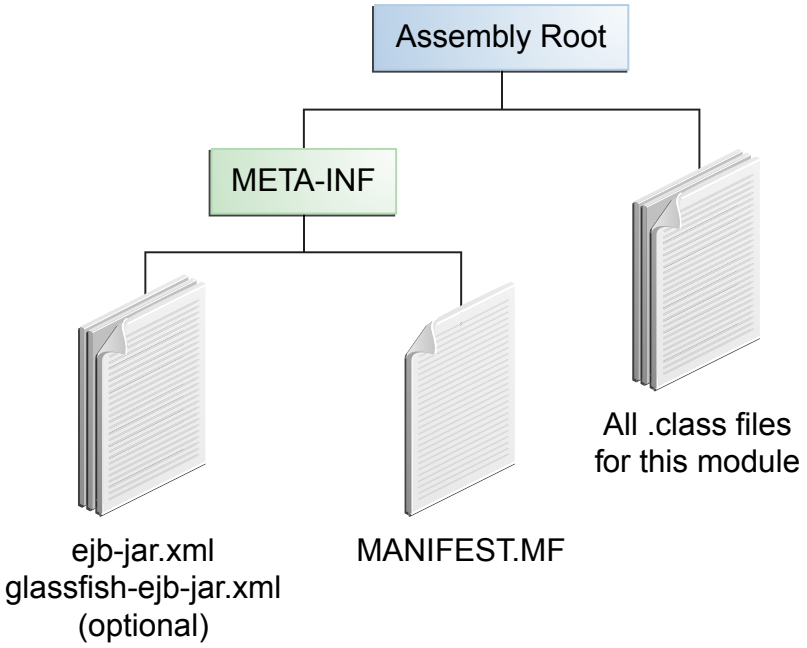


Abbildung 2.1: Struktur einer `jar`-Datei am Beispiel eines `ejb-jar`-Pakets aus [Ora14a].

Abbildung 2.1 zeigt den Aufbau eines `jar`-Archivs am Beispiel eines EJB Moduls. Assembly Root stellt das Wurzelverzeichnis des Archivs dar. Dort existieren neben dem Ordner `META-INF` noch die kompilierten Java Klassen für diese Komponente. Im Verzeichnis `META-INF` liegt das für ein `jar`-Archiv obligatorische `MANIFEST.MF` und den für diese EJB Komponente verfügbaren Deployment Deskriptor `ejb-jar.xml`.

2.2.2.2 war-Archiv

Eine `war`-Datei ist ein `jar`-Container, der für Webanwendungen gedacht ist. Die Verzeichnisstruktur eines `war`-Archivs kann in Abbildung 2.2 gesehen werden. Das hier gezeigte Beispiel zeigt den Java EE 7 Standard für eine GlassFish-Webanwendung.

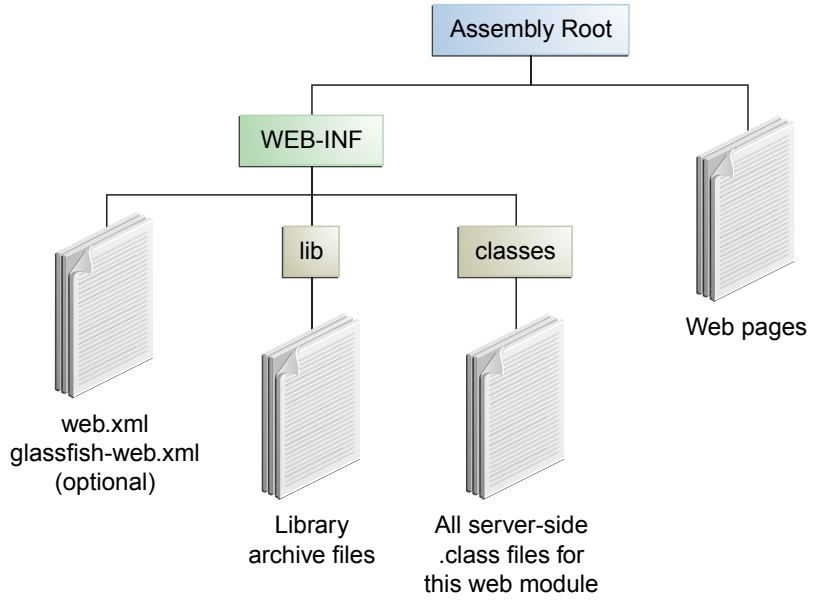


Abbildung 2.2: Ordnerstruktur eines `war`-Pakets aus [Ora14a].

Wie in Abbildung 2.2 zu sehen ist, befindet sich im `war`-Paket einer Java Webapplikation auf unterster Ebene das Verzeichnis `WEB-INF` und die Webseiten, die von der Anwendung ausgeliefert werden. Das können `jsp`-Dateien oder statischer Inhalt sein. Im Verzeichnis `WEB-INF` befindet sich der essenzielle `web.xml` Deployment Deskriptor (vgl. Kapitel 2.2.3.1). Zusätzlich ist hier noch mit `glassfish-web.xml` ein für den GlassFish Server (vgl. Abschnitt 2.3.1) spezifischen Deployment Deskriptor abgebildet, der im Standard nicht vorkommt. Im Verzeichnis `lib` befinden sich alle Java-Bibliotheken, die von der Anwendung benötigt werden, also alle in `jar`-Pakete zusammengefasste, kompilierte Klassen. Das Verzeichnis `classes` beinhaltet kompilierte Klassen, die nicht in `jar`-Paketen vorhanden sind, und andere Elemente, wie Konfigurationen, die zwangsläufig auf dem `classpath` liegen müssen.

2.2.2.3 rar-Archiv

In einer `rar`-Datei, also einem *Resource Adapter Archive*, werden XML-Dateien, Java Klassen und andere Objekte für Java EE Connector Architecture (JCA) Anwendungen hinterlegt. Das `rar`-Paket kann separat oder als Teil eines `ear`-Archivs auf eine Java EE Server installiert werden. Ebenso wie eine `war`-Datei ist das `rar`-Archiv von `jar` und damit `zip` abgeleitet. Ein Ressourcen Adapter Modul ist für alle anderen Anwendungen auf dem Java EE Server verfügbar und kann über eine Suchprozedur gefunden werden. [Ora14a]

2.2.2.4 ear-Archiv

Die Struktur eines `ear`-Pakets ist in Abbildung 2.3 dargestellt. Auf unterster Ebene der Ordnerstruktur befindet sich das Verzeichnis `META-INF`, in dem sich der Deployment Deskriptor `application.xml` (vgl. Kapitel 2.2.3.2) und ein serverspezifischer Deployment Deskriptor (in Abbildung 2.3 `glassfish-application.xml` für den Oracle Glassfish Server – vgl. Abschnitt 2.3.1) befinden können. Konfigurationen in den Deployment Deskriptoren überschrei-

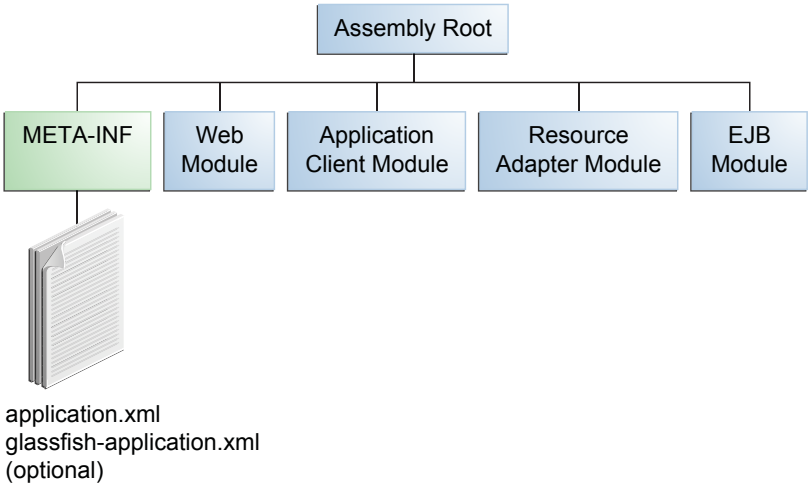


Abbildung 2.3: Ordnerstruktur eines `ear`-Pakets aus [Ora14a].

ben die Einstellungen im Code allerdings, wenn sie vorhanden sind. Daneben gibt es ein Web Modul, ein Applikationsclient Modul, ein Ressourcen Adapter Modul und ein EJB Modul.
Wie in Abbildung 2.4 gezeigt, können das EJB und das Web Modul mehrere einzelne Elemente enthalten. Das gilt auch für das Ressourcen Adapter und Anwendungsclient Modul, was allerdings nicht üblich ist. Die einzelnen Elemente können sich in separaten Ordnern im `ear`-Paket befinden, müssen aber nicht. Für mehr Übersicht bietet es sich an, die Elemente in einem für das zugehörende Modul eigenen Verzeichnis zu gruppieren. Hierbei können alle Komponenten vom selben Modultyp zusätzlich mit einem eigenen Deployment Deskriptor versehen werden. [Sun09]
Sollen alle Komponenten der Java EE Anwendung gemeinsame Bibliotheken verwenden, so können diese im optionalen Verzeichnis `lib` auf unterster Ebene des `ear`-Pakets abgelegt werden. [DS13b]
Wenn kein Deployment Deskriptor `META-INF/application.xml` vorhanden ist, werden die Modultypen beim Deployment auf einem Applikationsserver nach folgendem Schema identifiziert: [DS13b]

1. Alle Dateien im `ear`-Paket mit der Dateiendung `war` werden als Web Modul angesehen.

2. Alle Dateien im `ear`-Paket mit der Dateiendung `rar` werden als Ressourcen Adapter Modul angesehen.

3. Falls das Verzeichnis `lib` im `ear`-Paket vorhanden ist, wird es Verzeichnis für gemeinsame Bibliotheken verwendet.

4. Alle Daiten im `ear`-Paket, die nicht im `lib`-Verzeichnis liegen, werden folgendermaßen klassifiziert:

   a) Wenn im Manifest (`META-INF/MANIFEST.MF`) der `jar`-Datei das Attribut `Main-Class` vergeben ist oder der Applikationsclient Deployment Deskriptor (`META-INF/application-client.xml`) vorhanden ist, wird diese Komponente als Applikationsclient Modul klassifiziert.

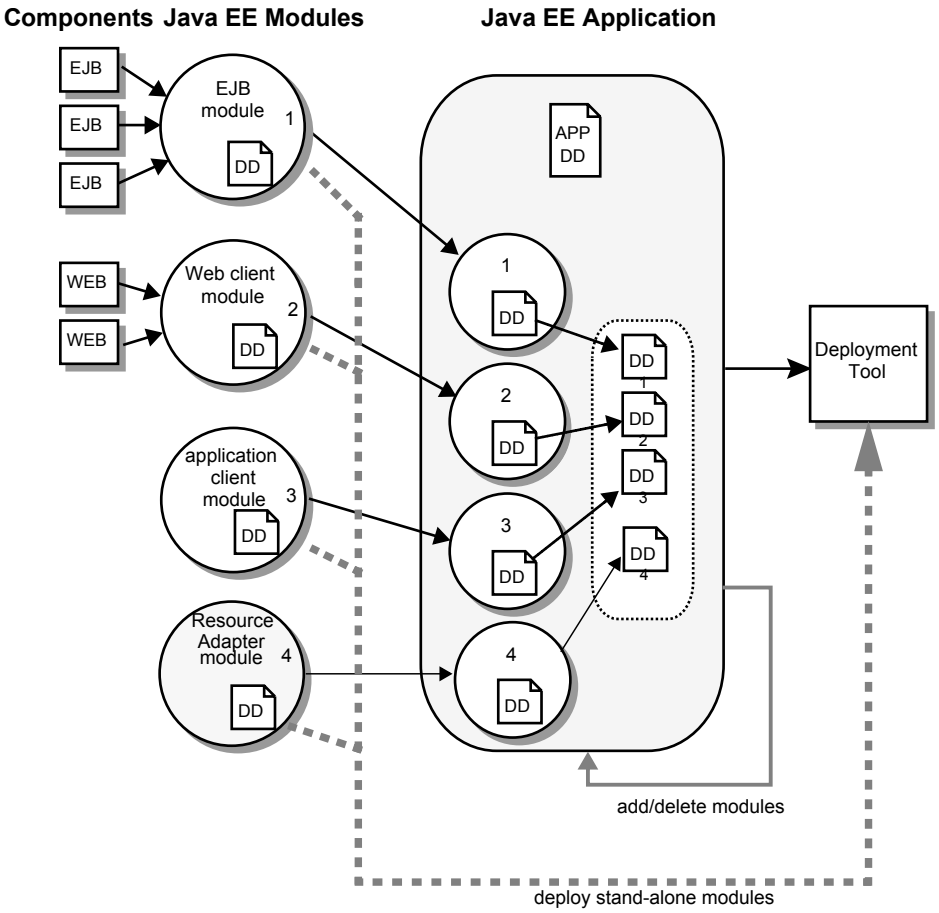


Abbildung 2.4: Struktur eines `ear`-Pakets mit Deploymentinformationen aus [DS13b, Figure EE.8-1].

b) Wenn das `jar`-Paket einen EJB Deployment Deskriptor (`META-INF/ejb-jar.xml`) enthält oder über den Anwendungscode als EJB erkennbar ist, wird diese Komponente als EJB Modul identifiziert.

c) Alle anderen `jar`-Dateien werden ignoriert solange sie nicht von einer klassifizierten Komponente referenziert werden.

2.2.3 Deployment Deskriptoren

Deployment Deskriptoren sind Beschreibungen von JAVA Web-Applikationen, die im Applikationspaket ausgeliefert werden und Klassen, Ressourcen und Konfigurationen beinhalten, um dem Applikationsserver mitzuteilen wie Anfragen bearbeitet werden sollen. Bei einer Anfrage an den Server wird die URL mit Hilfe der Deployment Deskriptoren an die richtige Codestelle weitergeleitet. [Sal14]
Die hier beschriebenen Deployment Deskriptoren sind `web.xml` und `application.xml`. Die in diesen beiden Deskriptoren möglichen Einstellungen sollten unabhängig vom Applikationsserver verwendeten werden können. Es gibt noch unzählige weitere Deployment Deskriptoren wie `ra.xml` für Ressourcenadapter Module, `ejb-jar.xml` für EJB Module, `application-client.xml` für Anwendungsclient Module oder `faces-config.xml` für Seitennavigation in

Webanwendungen, auf die hier nicht explizit eingegangen wird. Zusätzlich existieren für die meisten Applikationsserver noch optionale individuelle Deployment Deskriptoren, die zusätzliche Einstellungen erlauben. Auf diese wird hier verzichtet, da die Konfiguration immer ähnlich verläuft und hier nur Beispiele gegeben werden, um den Mechanismus kurz zu beleuchten. [Sal14]
Alle möglichen Parameter für die hier vorgestellten Deployment Deskriptoren können zusammengefasst unter [Orad] gefunden werden.

2.2.3.1 web.xml

Die `web.xml` befindet sich im Verzeichnis `WEB-INF` eines war-Pakets und ist der wichtigste Deployment Deskriptor von Java Webapplikationen. In der `web.xml` wird im Element `web-app` die zu konfigurierende Webapplikation definiert und beschrieben. Dieses Element definiert auch die Version und damit das entsprechende XSD, mit dem die möglichen Parameter definiert werden. Eine sehr umfassende Übersicht über alle möglichen Einstellungen und Versionen der `web.xml` findet sich beispielsweise online bei [Met]. Das folgende Beispiel zeigt eine `web.xml` für Java EE 7, also Servlet 3.1: [Met; Sal14]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">
</web-app>
```

Die Version der `web.xml` ist auch gleichzeitig die Java Servlet Version und steht somit in Verbindung mit der Java EE Version. Eine Übersicht über die möglichen Servlet Versionen und deren Erscheinen bietet Tabelle 2.2.

Tabelle 2.2: Übersicht über die Servlet Versionen [Chh].

Java EE Version	Servlet Version	Veröffentlichung
J2EE 1.2	Servlet 2.2	12.12.1999
J2EE 1.3	Servlet 2.3	24.09.2001
J2EE 1.4	Servlet 2.4	11.11.2003
Java EE 5	Servlet 2.5	11.05.2006
Java EE 6	Servlet 3.0	10.12.2009
Java EE 7	Servlet 3.1	12.06.2013

Elemente der `web.xml` sind in der Servlet Spezifikation zu finden [CM13]. Die Syntax der `web.xml` ist normale XML-Syntax, wobei das Wurzelelement `web-app` darstellt. In diesem Knoten werden dann alle anderen Elemente definiert. In der `web.xml` sind die folgenden Konfigurationstypen und Deploymentinformationen enthalten: [CM13]

- Initialisierungsparameter des `ServletContext`
- Sitzungskonfiguration

- Servlet-JSP Definitionen
- Servlet-JSP Abbildungen
- Anwendungslebenszyklus Listener Klassen
- Filter Definitionen
- Filter Abbildungen
- MIME Typen Abbildungen
- Willkommen Dateiliste
- Fehlerseiten
- Sprachumgebung und Kodierung
- Sicherheit

2.2.3.2 application.xml

Der Deployment Deskriptor `application.xml` muss im Verzeichnis `META-INF` eines `ear`-Pakets liegen, also einer Java-EE-Anwendung. Hierin werden die einzelnen Applikationen in diesem Paket definiert und konfiguriert. Die einzelnen Bestandteile eines `ear`-Archivs können in Kapitel 2.2.2.4 nachgelesen werden. [DS13b]

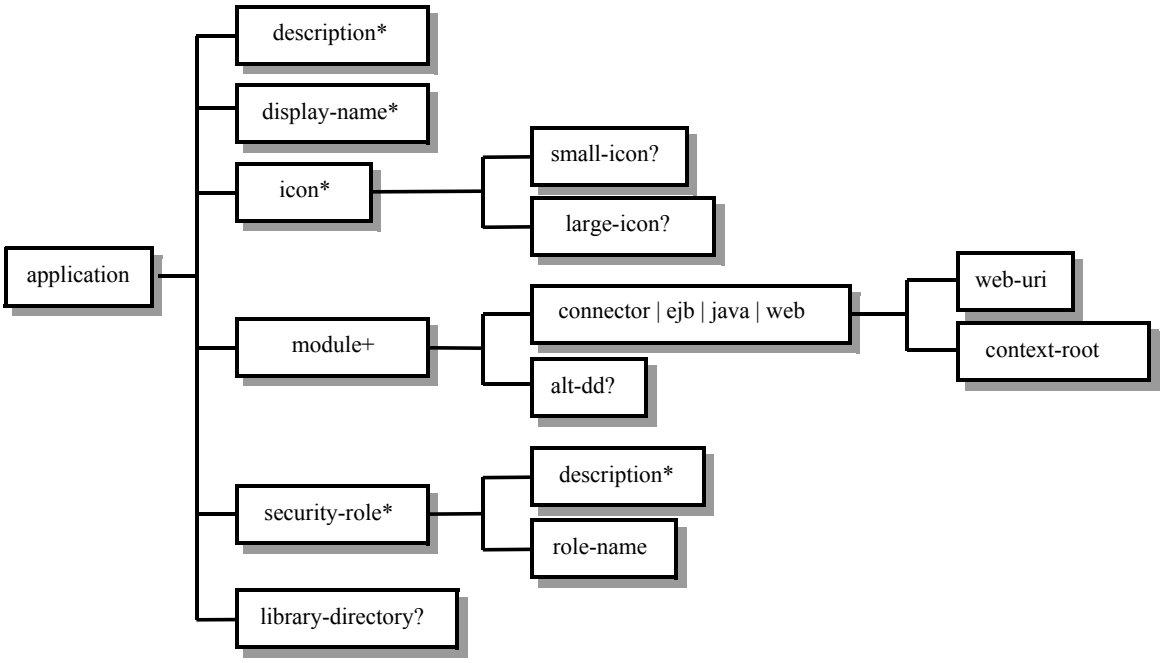


Abbildung 2.5: Die Elemente einer `application.xml` einer Enterprise Anwendung aus [DS13b, Figure EE.A-3] für Java EE 5. Hierbei bedeutet $*$, dass das Element beliebig oft, + mindestens einmal und ? null oder einmal vorkommen kann.

Abbildung 2.5 zeigt die Struktur einer `application.xml` am Beispiel von Java EE 5. In den Versionen 6 und 7 sind viele zusätzliche Elemente zu `application.xml` hinzugekommen, wodurch die Darstellung der hier gezeigten wichtigsten Elemente nicht übersichtlich möglich ist. Das äußerste Element des XML Schemas einer `application.xml` heißt `<application>`, das im folgenden Beispiel gezeigt wird [DS13b]:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/application_7.xsd"
    version="7">
</application>
```

Die in Abbildung 2.5 gezeigten Elemente sind in der folgenden Auflistung genauer beschrieben:

- `<description>` Häufigkeit: $0 - \infty$

  Eine Beschreibung der Enterprise Anwendung.

- `<display-name>` Häufigkeit: $0 - \infty$

  Der Anzeigename der Enterprise Anwendung.

- `<icon>` Häufigkeit: $0 - 1$

  Die Speicherorte des angezeigten Bilds der Enterprise Anwendung mit den Subelementen:

  - `<small-icon>` Häufigkeit: $0 - 1$

    Optionaler Parameter für ein niedrig aufgelöstes Bild der Anwendung.

  - `<large-icon>` Häufigkeit: $0 - 1$

    Optionaler Parameter für ein höher aufgelöstes Bild der Anwendung.

- `<module>` Häufigkeit: $1 - \infty$

  Für jedes in der Enterprise Anwendung vorhandene Modul muss in der `application.xml` ein `<module>` Element enthalten sein mit genau einem der folgenden Subelemente:

  - `<connector>`

    Mit diesem Parameter wird ein Ressourcenadapter definiert, der der JCA genügt. Hier ein Beispiel für den Ressourcenadapter `beispiel-rar.rar`:

    `<connector>beispiel-rar.rar</connector>`

  - `<ejb>`

    Dieser Parameter definiert ein EJB Modul wie beispielsweise `beispiel-ejb.jar`:

    `<ejb>beispiel-ejb.jar</ejb>`

- `<java>`

  Mit dem Element `java` wird ein Anwendungsclient Modul definiert. Hier im Beispiel `beispiel-client.jar`:

  ```
  <java>beispiel-client.jar</java>
  ```

- `<web>`

  Definiert eine Webanwendung, wobei die folgenden Subelemente notwendig sind:

  - `<web-uri>` Häufigkeit: 1

    Definiert den Pfad zum `war`-Archiv der Webanwendung.

  - `<context-root>` Häufigkeit: 1

    Definiert den URL-Pfad zur Webanwendung.

  Beispielhaft für eine Anwendung `beispiel-web.war` mit URL-Pfad `beispiel-web` wäre der Eintrag:

  ```
  <web>
      <web-uri>beispiel-web.war</web-uri>
      <context-root>beispiel-web</context-root>
  </web>
  ```

Zusätzlich kann noch der optionale zusätzliche Parameter `alt-dd` definiert werden, der alternative Deployment Deskriptoren definiert, um diese zu laden.

- `<security-role>` Häufigkeit: $0 - \infty$

  Mit einer `<security-role>` werden globale Sicherheitsrollen für die Anwendung definiert. Subelemente sind:

  - `<description>` Häufigkeit: $0 - 1$

    Die optionale Beschreibung eine Sicherheitsrolle.

  - `<role-name>` Häufigkeit: 1

    Dieser Parameter definiert den Namen der Sicherheitsrolle und wird für die Authentifizierung innerhalb der Anwendung verwendet.

  Ein Beispiel für eine Sicherheitsrolle für normale Anwender wäre:

  ```
  <security-role>
      <description>Sicherheitsrolle für normale Anwender</description>
      <role-name>Anwender</role-name>
  </security-role>
  ```

- `<library-directory>` Häufigkeit: $0 - 1$

  Mit diesem optionalen Parameter wird das zusätzliche Verzeichnis in den Pfad zur Suche nach Klassen und anderen Dateien aufgenommen, das hier definiert wird.

2.3 Java EE Applikationsserver

In diesem Kapitel wird auf die folgenden Java EE Applikationsserver eingegangen:

- Oracle GlassFish (Abschnitt 2.3.1)
- Oracle WebLogic (Abschnitt 2.3.2)
- IBM WebSphere (Abschnitt 2.3.3)
- RedHat WildFly (Abschnitt 2.3.4)
- Apache Tomcat (Abschnitt 2.3.5)

Hierbei soll ein Überblick über die Entstehung der Anwendungsserver und deren Bestandteile gegeben werden. Zusätzlich werden Unterschiede zum Java EE Standard aufgezeigt, die im Normalfall auch die Server untereinander unterscheiden. Dazu gehören hauptsächlich individuelle Deployment Deskriptoren.
Generell gibt es wie in Kapitel 2.1 angesprochen im Wesentlichen zwei Arten von Java EE Servern, nämlich die Server, die das Full Profile implementieren, und die, die nur das Web Profile implementieren. Das Full Profile ist eine komplette Implementierung der im Java EE Standard enthaltenen APIs, wohingegen das Web Profile nur die für Web Anwendungen relevanten APIs beinhaltet. [Sal14]

2.3.1 Oracle GlassFish

Oracle GlassFish Server ist die kommerzielle Version des *GlassFish Server Open Source Edition* und geht auf den *Sun GlassFish Enterprise Server* zurück, der wiederum auf dem *Sun Java System Application Server* basiert (siehe Logo in Abbildung 2.6). GlassFish ist ein Java EE Server, der mittlerweile in Version 4 verfügbar ist und in den Versionen 3 und 4 die Referenzimplementierung eines Java EE Applikationsservers für Java EE 6 und 7 respektive war. 2013 gab Oracle bekannt, dass es keine weiteren kommerziellen Versionen mehr geben werde. Kommerzielle Kunden werden auf den *Oracle WebLogic Server* (siehe Abschnitt 2.3.2) verwiesen. Daneben gibt es eine große Gemeinschaft, die den Server open-source weiterentwickelt, wobei geplant ist mit dem *GlassFish Server Open Source Edition 5* die Referenzimplementierung des noch nicht erschienenen Java EE 8 zu erreichen. [Ora10; Ora13a; Orab]

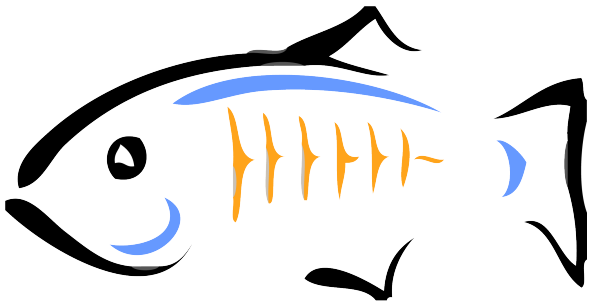


Abbildung 2.6: Das Logo des GlassFish Applikationsservers von [Oraa].

Die wichtigsten Bestandteile von GlassFish sind [Ora13a]:

Apache Felix

Ein Framework, das das Deployment von Anwendungen durchführt.

TopLink

TopLink ist eine Java Persistence API Implementierung von Oracle, deren Weiterentwicklung *EclipseLink* von GlassFish verwendet wird.

Grizzly

Ein http-Frontend, das auf eine Abwandlung von *Apache Tomcat* aufgesetzt ist und *Java New I/O* (NIO) aus Geschwindigkeitsgründen verwendet.

Als Referenzimplementierung für Java EE 7 beinhaltet GlassFish 4 das Full Profile von Java EE 7. Zusätzlich zu den gebräuchlichen und im Java EE Standard enthaltenen Deployment Deskriptoren unterstützt GlassFish (hier als Beispiel in Version 4) noch folgende zusätzliche, individuelle Deployment Deskriptoren: [Ora13b]

- `glassfish-application.xml`

  Die Datei `glassfish-application.xml` konfiguriert eine komplette Java EE Anwendung (`ear`-Datei). Das äußerste Element heißt `<glassfish-application>`. Dieser Deployment Deskriptor kann verwendet werden, um die normale `application.xml` zu ersetzen oder zu erweitern.

- `glassfish-web.xml`

  Die Datei `glassfish-web.xml` konfiguriert eine komplette Webanwendung (`war`-Datei) und kann eine normale `web.xml` ersetzen oder erweitern. Das äußerste Element hat den Namen `<glassfish-web-app>`.

- `glassfish-ejb-jar.xml`

  Mit `glassfish-ejb-jar.xml` kann eine kompletter Enterprise Bean (EJB `jar`-Datei) konfiguriert werden. Das äußerste Element hat die Bezeichnung `<glassfish-ejb-jar>`. Der normale Deployment Deskriptor `ejb-jar.xml` kann mit diesem ersetzt oder erweitert werden.

- `sun-cmp-mappings.xml`

  Mit der Datei `sun-cmp-mappings.xml` kann eine von einem Container verwaltete Persistenz für ein EJB konfiguriert werden. Für diesen Deployment Deskriptor existiert kein Äquivalent im Java EE Standard; er erweitert den normalen `ejb-jar.xml` Deployment Deskriptor. Das äußerste Element hat die Bezeichnung `<sun-cmp-mappings>`.

- `glassfish-application-client.xml`

  Mit dem Deployment Deskriptor `glassfish-application-client.xml` kann `application-client.xml` erweitert oder ersetzt werden. Er konfiguriert einen Anwendungsclient Container (`jar`-Datei) und hat als äußerstes Element `<glassfish-application-client>`.

- `sun-acc.xml`

  Mit der Datei `sun-acc.xml` kann ein Anwendungsclient Container konfiguriert werden. Diese Datei ist kein wirklich Deployment Deskriptor, sondern eher eine Konfigurationsdatei und kann die im GlassFish Server vorhandene `sun-acc.xml` ersetzen. Das äußerste Element hat den Namen `<client-container>`.

- `glassfish-resources.xml`

  Mit der Datei `glassfish-resources.xml` können Ressourcen konfiguriert werden, die für die einzelne Anwendung erreichbar sein sollen. Zusätzlich können die in dieser Datei konfigurierten Ressourcen auch direkt in den Server geladen und auf diese Weise alle verfügbaren Anwendungen auf dem Server zur Verfügung stehen. Der Deployment Deskriptor hat kein Äquivalent im Java EE Standard und besitzt das äußerste Element `<resources>`.

Aus Gründen der Kompatibilität zu älteren Versionen werden zusätzlich die älteren Deployment Deskriptoren `sun-application.xml`, `sun-web.xml`, `sun-ejb-jar.xml`, `sun-application-client.xml` und `sun-resources.xml` weiter unterstützt. Eine Beschreibung aller in den GlassFish spezifischen Deployment Deskriptoren möglichen Elemente ist in [Ora13b, C - Elements of the GlassFish Server Deployment Descriptors] nachzulesen.
Neben diesen individuellen Deployment Deskriptoren werden auch manche Elemente aus WebLogic spezifischen Deployment Deskriptoren unterstützt: [Ora13b]

- `weblogic-application.xml`

  Das einzige unterstützte Element aus der `weblogic-appplication.xml` ist `<security>` und besitzt das Äquivalent `<security-role-mapping>` in der `glassfish-application.xml`.

- `weblogic.xml`

  Es werden einige Elemente aus der `weblogic.xml` von GlassFish unterstützt. Details können in [Ora13b, Table B-2] nachgelesen werden.

- `weblogic-webservices.xml`

  Manche der Elemente aus `weblogic-webservices.xml` werden von GlassFish unterstützt. Genaue Informationen sind in [Ora13b, Table B-3] zu finden.

2.3.2 Oracle WebLogic

Der *Oracle WebLogic Server* ist ein sehr umfangreicher und weit verbreiteter Java EE Applikationsserver (Das Logo ist in Abbildung 2.7 zu sehen). Er war auch der erste Anwendungsserver, der einen Java EE Standard komplett implementierte. Bis zur Version 10 (im Jahr 2008) war er unter dem Namen BEA WebLogic bekannt. Anschließend wurde die Firma BEA von Oracle gekauft und der Server in Oracle WebLogic Server umbenannt. Nach [Web] sind die Lizenzkosten eines Oracle WebLogic Servers die höchsten auf dem Markt. Das Produkt ist aber dennoch weit verbreitet. Die aktuelle Version 12c Release 3 des Oracle WebLogic Servers implementiert das Full Profile von Java EE 6 und einzelne APIs von Java EE 7. Zu diesen Java EE 7 APIs gehören JPA 2.1, JAX-RS 2.0, JSON-P 1.0 und WebSockets 1.0. [Jam10; Ora13c; Ora14b; Ora14c]

Abbildung 2.7: Das Logo des Oracle WebLogic Servers in Version 12c Release 3 aus [Ora14c].

Der Oracle WebLogic Server ist Teil der *Oracle Fusion Middleware*. Middleware ist die Software, die einzelne Programmkomponenten oder Enterprise Anwendungen untereinander verbindet. Sie bildet eine Schicht, die zwischen Betriebssystem und verteilten Anwendungen liegt. Dazu gehören Web Server, Anwendungsserver, Verwaltungssysteme für Informationen und Systeme, die Anwendungsentwicklung und -bereitstellung unterstützen. [Jam10]
Die Kernkomponenten von WebLogic sind: [Jam10; Ora13c]

Oracle WebLogic Server Webservices

Die *Oracle WebLogic Server Webservices* sind eine Implementierung der Webservices API des Java EE Standards.

Oracle Coherence

Oracle Coherence ist ein weiterer Bestandteil der *Oracle Fusion Middleware* und ist eine Schnittstelle, um regelmäßig verwendete Daten schnell und zuverlässig zu laden.

JMS Messaging Standard

Neben dem normalen *Java Messaging Service* (JMS) stellt WebLogic eine Erweiterung zur Verfügung, die besseres Queuing ermöglicht.

Enterprise Grid Messaging

Oracle WebLogic Server Enterprise Grid Messaging ist ein vollwertiges Unternehmensnachrichtensystem, das als Teil des Anwendungsservers auf die gleichen Ressourcen zugreift und damit Mehraufwand reduziert.

JRockit

JRockit ist eine hochleistungsfähige Erweiterung der normalen JVM (*Java Virtual Machine*), die Java Anwendungen sicherer, zuverlässiger, skalierbarer und flexibler machen soll.

Tuxedo

Tuxedo ist eine skalierbare, vielsprachige und hochleistungsfähige Plattform zur Übermittlung von Nachrichten und Durchführung von Transaktionen auf verteilen Umgebungen.

Oracle TopLink

Oracle TopLink ist eine leistungsfähige Implementierung der Java Persistence API des Java EE Standards, die ein Framework zur Verfügung stellt, um Objektpersistenz und -transformation zu erreichen.

Für den WebLogic Anwendungsserver gibt es aufgrund von Eigenentwicklungen und Erweiterungen des Java EE Standards individuelle Deployment Deskriptoren, die nachfolgend kurz für die Version 12c Release 3 beschrieben sind: [Ora15e]

- `weblogic.xml`

  Die `weblogic.xml` Datei ist ein Ersatz oder eine Erweiterung der Deployment Deskriptoren `web.xml` und `WEB-INF/beans.xml` von Webanwendungen und besitzt das äußerste Element `<weblogic-web-app>`.

- `weblogic-ejb-jar.xml` **und** `weblogic-cmp-rdbms-jar.xml`

  Der Deployment Deskriptor `weblogic-ejb-jar.xml` erweitert oder ersetzt `ejb-jar.xml` eines EJB Moduls. Zusätzlich kann der Deskriptor `weblogic-cmp-rdbms-jar.xml` diesen noch für EJB 2.1 erweitern. Die äußersten Elemente von `weblogic-cmp-rdbms-jar.xml` und `weblogic-ejb-jar.xml` der zugehörigen XML Schemata sind `<weblogic-rdbms-jar>` und `<weblogic-ejb-jar>`.

- `weblogic-rdbms-jar.xml` **und** `persistence-configuration.xml`

  Die beiden Deployment Deskriptoren `weblogic-rdbms-jar.xml` und `persistence-configuration.xml` ersetzen oder erweitern den Deployment Descriptor `META-INF/beans.xml` eines EJB Moduls. Die äußersten Elemente im XML Schema der beiden Dateien sind `<weblogic-rdbms-jar>` und `<persistence-configuration>`.

- `weblogic-webservices.xml, weblogic-wsee-clientHandlerChain.xml, weblogic-webservices-policy.xml` **und** `weblogic-wsee-standaloneclient.xml`

  Die Deployment Deskriptoren `weblogic-webservices.xml`, `weblogic-wsee-clientHandlerChain.xml`, `weblogic-webservices-policy.xml` und `weblogic-wsee-standaloneclient.xml` können den Java EE Deployment Deskriptor `webservices.xml` komplett ersetzen oder erweitern. Hier sieht man, dass ein sehr hoher Stellenwert bei WebLogic auf Webservices gelegt wird. Diese können umfassend konfiguriert werden. Die äußersten Elemente in den entsprechenden XML Schemata haben die Bezeichnungen `<weblogic-webservices>`, `<weblogic-wsee-clientHandlerChain>`, `<webservice-policy-ref>` und `<weblogic-wsee-standaloneclient>`.

- `weblogic-ra.xml`

  Der Deployment Deskriptor `weblogic-ra.xml` konfiguriert ein Ressourcenadapter Modul und ersetzt oder erweitert die entsprechenden `ra.xml` und `META-INF/beans.xml`. Das äußerste Element des XML Schemas heißt `<weblogic-connector>`.

- `weblogic-application.xml`

  Mit dem Deployment Deskriptor `weblogic-application.xml` können Enterprise Anwendungen konfiguriert werden und somit der Java EE Deployment Deskriptor `application.xml` ersetzt oder erweitert werden. Das äußerste Element im zugehörigen XML Schema hat den Namen `<weblogic-application>`.

- `«MODUL-NAME».runtime.xml`

  Für Anwendungsclient Module ist der Weblogic spezifische Deployment Deskriptor eine XML Datei, die zur Laufzeit ausgewertet wird und liegt parallel zum Modul mit dem

Namen «MODUL-NAME».runtime.xml. Sie ersetzt oder erweitert die Datei application-client.xml. Das Wurzelelement heißt wie bei der application-client.xml <application-client>.

Zusätzlich zu diesen Deployment Deskriptoren, die vorhandene Deployment Deskriptoren aus dem Java EE Standard ersetzen oder erweitern, besitzt WebLogic noch eigene, die über den Funktionsumfang der Standard Deployment Deskriptoren hinausgehen und komplett neue Elemente mitbringen: [Ora15e]

- weblogic-pubsub.xml

  Mit dem Deployment Deskriptor weblogic-pubsub.xml können *HTTP Publish-Subscribe Server* konfiguriert werden. Das ist ein Mechanismus, bei dem Webclients Kanäle abonnieren und dann Nachrichten über ein asynchrones Nachrichtensystem mittels HTTP publizieren. WebLogic enthält einen eigenen HTTP Publish-Subscribe Server, der mit diesem Deployment Deskriptor beschrieben und konfiguriert werden kann. Das Wurzelelement des zugehörigen Schemas heißt <wlps:weblogic-pubsub> und die Datei liegt im Ordner WEB-INF der Webanwendung. [Ora15f]

- «DATEI-NAME»-jms.xml

  Mit diesem Deployment Deskriptor können JMS Ressourcen konfiguriert werden, wobei <DATEI-NAME> jeder beliebige Name sein kann. Das äußerste Element heißt <weblogic-jms> und das entsprechende Modul muss in WebLogic mit Pfadangabe definiert sein. [Ora15b]

- «DATEI-NAME»-jdbc.xml

  Mit diesem Deployment Deskriptor können JDBC Ressourcen, also Datenbankverbindungen, konfiguriert werden. Das äußerste Element heißt <jdbc-data-source> und das entsprechende Modul muss in WebLogic als JDBC Ressource definiert sein. [Ora15a]

- plan.xml

  Mit dem Deployment Deskriptor plan.xml kann die Reihenfolge und die Konfiguration des Deployments von Anwendungen auf einem WebLogic Server konfiguriert werden. Das Wurzelelement des zugehörigen XML Schemas ist <deployment-plan>. [Ora15d]

- weblogic-diagnostics.xml

  Mit dem Deskriptor weblogic-diagnostics.xml kann das *WebLogic Diagnostics Framework* beschrieben und konfiguriert werden. Dieses Framework beinhaltet Komponenten zum Sammeln und Auswerten von Daten wie Leistungsmessungen, Überwachungsdaten und Logdateien. Das Wurzelelement des zugehörigen XML Schemas heißt <wldf-resource>. [Ora15c]

- coherence-application.xml

  Coherence Applications sind Module, die auf einem WebLogic Server installiert werden können. Hierbei steht *Coherence* für eine Softwarelösung, die verteiltes Caching und Grid-Computing ermöglichen soll. Anwendungen nutzen das, um die Skalierbarkeit, Sicherheit und Verfügbarkeit zu verbessern. Um eine solche Anwendung zu konfigurieren,

kann mit dem Deployment Deskriptor `coherence-application.xml` gearbeitet werden. Diese Datei befindet sich im Ordner `META-INF` des zugehörigen `jar`-Pakets und besitzt im XML Schema das äußerste Element `<coherence-application>`. [Ora14d]

Als Anwendungsserver von Oracle unterstützt WebLogic auch GlassFish spezifischen Deployment Deskriptoren. In diesem Fall werden nur die Deskriptoren `glassfish-web.xml` und `sub-web.xml` unterstützt, wobei diese ignoriert werden, sobald eine `weblogic.xml` vorhanden ist. Das gleiche gilt auch für die `sun-web.xml` wenn `glassfish-web.xml` gefunden wird. Werden Einstellungen aus einem dieser Deployment Deskriptoren verwendet oder ignoriert, so wird beim Serverstart eine entsprechende Information ausgegeben. [Ora15f]

2.3.3 IBM WebSphere

IBM WebSphere ist der kommerzielle Java EE Anwendungsserver der Firma IBM (Logo in Abbildung 2.8). Die erste Version wurde 1998 fertig gestellt und basiert auf dem *Apache* Webserver. Die aktuelle Version 8.5.5 implementiert das Full Profile von Java EE 6 und ist 2013 erschienen. Neben dem normalen WebSphere Applikationsserver existiert auch noch das *Liberty Profile*, das als eigenständiger Server verwendet werden kann (*WebSphere Application Server Liberty Core*), aber auch Teil der normalen Distribution ist. Das Liberty Profile implementiert nicht den kompletten Java EE Standard, sondern nur das Web Profil und stellt eine leichtgewichtige und vereinfachte Form des WebSphere Servers dar. Hiermit sollen Webanwendungen unterstützt und einfacher konfiguriert werden können als bei anderen Java EE Anwendungsservern. Die genaue Liste der von Liberty unterstützten Java EE 6 APIs ist beispielsweise in [ABB+13, Table 1-2] zu finden. [IBMc; ABB+13]

Abbildung 2.8: Das Logo des IBM WebSphere Servers in Version 8 aus [IBM13].

Einige Merkmale, die WebSphere von anderen Anwendungsservern abheben, sind: [Web]

- In einem Cluster können unterschiedliche Version von WebSphere gleichzeitig arbeiten und überwacht werden.
- Die Überwachung von Anwendungen und deren Verbesserung durch serverseitige Einstellungen ist in WebSphere integriert und kann zur Laufzeit angepasst werden.
- WebSphere beinhaltet die Möglichkeit *Portlets* auszuführen. Bei anderen Applikationsservern muss ein Modul hierfür nachinstalliert werden, das zusätzliche Kosten mit sich bringt.
- Bei WebSphere können theoretisch unendlich viele Versionen der gleichen Anwendung gleichzeitig auf einem Server ausgeführt werden.

Neben den gängigen Java EE Deployment Deskriptoren besitzt WebSphere drei eigene individuelle Arten von Deployment Deskriptoren. Diese haben die Dateinamen `ibm-xxx-bnd.xml`,

`ibm-xxx-ext.xml` und `ibm-xxx-ext-pme.xml`, wobei bis Java EE 5 die Endung `xmi` verwendet wurde. Hierbei steht `xxx` für die Art der zu konfigurierenden Applikation. Die `ibm-xxx-bnd.xml` Deskriptoren werden verwendet, um externe Referenzen auf andere Ressourcen zu definieren. Mit `ibm-xxx-ext.xml` werden WebSphere spezifische Erweiterungen zum vorhandenen Deployment Deskriptor eingeführt. Zusätzlich können mit `ibm-xxx-ext-pme.xml` WebSphere Erweiterungen am Programmiermodel des Java EE Standards vorgenommen werden. Hier werden die Deployment Deskriptoren angesprochen und kurz auf deren Schema eingegangen: [ABB+12]

- `ibm-application-bnd.xml`, `ibm-application-ext.xml` **und** `ibm-application-ext-pme.xml`

  Mit `ibm-application-bnd.xml`, `ibm-application-ext.xml` und `ibm-application-ext-pme.xml` kann eine Enterprise Applikation neben der `application.xml` weiter konfiguriert werden. Die Wurzelelemente der entsprechenden XML Schemata heißen `<application-bnd>`, `<application-ext>` und `<application-ext-pme>`.

- `ibm-ejb-jar-bnd.xml`, `ibm-ejb-jar-ext.xml` **und** `ibm-ejb-jar-ext-pme.xml`

  Ein EJB Modul kann mit den Deskriptoren `ibm-ejb-jar-bnd.xml`, `ibm-ejb-jar-ext.xml` und `ibm-ejb-jar-ext-pme.xml` zusätzlich zur `ejb-jar.xml` weiter konfiguriert werden. Die Wurzelelemente der entsprechenden XML Schemata heißen `<ejb-jar-bnd>`, `<ejb-jar-ext>` und `<ejb-jar-ext-pme>`.

- `ibm-web-bnd.xml`, `ibm-web-ext.xml` **und** `ibm-web-ext-pme.xml`

  Die Dateien `ibm-web-bnd.xml`, `ibm-web-ext.xml` und `ibm-web-ext-pme.xml` können eine Webapplikation über den Umfang der `web.xml` hinaus konfigurieren. Die Wurzelelemente der entsprechenden XML Schemata heißen `<web-bnd>`, `<web-app-ext>` und `<web-app-ext-pme>`.

2.3.4 RedHat WildFly

Der Applikationsserver von Red Hat heißt *WildFly*, ist kostenlos und open-source. 2002 wurde dieser komplett in Java geschriebene Anwendungsserver unter dem Namen *JBoss Application Server* vorgestellt. Mit Version 8 wurde JBoss Application Server in WildFly umbenannt, wobei die Versionsnummern fortgeführt werden. Die aktuelle WildFly Version 9 implementiert das Full Profile von Java EE 7. Das offizielle Logo von WildFly ist in Abbildung 2.9 zu sehen. Die kommerzielle Version trägt weiterhin den Namen *JBoss Enterprise Application Plattform* und stellt eine leicht veränderte Version des älteren JBoss Application Servers dar. [Rede; Redc]

Abbildung 2.9: Das Logo von Red Hat WildFly aus [Redd].

Die Kernkomponenten von WildFly sind selbst open-source und wurden mehrheitlich von der *JBoss Community* entwickelt, die auch für WildFly verantwortlich ist: [Redf; Rede]

Hibernate

Hibernate ist eine leistungsfähige Java Persistence API (JPA) Implementierung.

Narayana

Narayana ist ein Transaktionsmanager, der früher auch als *JBossTS* und *Arjuna Transaction Service* bekannt war und seit 2005 von Red Hat/Jboss Community weiterentwickelt wird. Er erfüllt die Voraussetzungen für die Java EE API *JTA* (Java Transaction API). [Redg]

Infinispan

Infinispan ist eine Implementierung von *JMX* (Java Management Extensions) einem weiteren Teil des Java EE Standards. Zusätzlich stellt es ein über verschiedene System erweiterbares, sich im Arbeitsspeicher befindliches Datennetz dar.

IronJacamar

IronJacamar ist eine Implementierung der Java EE Connector Architecture (*JCA*). JCA definiert eine Standardarchitektur, um die Java EE Plattform mit heterogenen Enterprise Informationssystemen zu verbinden. [Redh]

RESTEasy

Die *JAX-RS* (Java API for RESTful Web Services) also RESTFul Webservices Implementierung von WildFly heißt *RESTEasy*. JAX-RS ist eine reine serverseitige Spezifikation, wohingegen RESTEasy auch am Client eingesetzt werden kann. [Redi]

Weld

Weld ist die Referenzimplementierung von *CDI*, der Contexts and Dependency Injection API des Java EE Standards. Weld wird neben WildFly auch in GlassFish, WebLogic, Tomcat oder Jetty eingesetzt. [KMH+]

HornetQ

HornetQ ist eine Implementierung von *JMS*, dem Java Messaging Standard. Es stellt ein sehr leistungsfähiges, geclustertes, asynchrones und gut einbettbares Nachrichtensystem dar, das viele verschiedene Protokolltypen unterstützt. [STF+]

Mojarra

Mojarra ist die Referenzimplementierung von *JSF*, den Java Server Faces und wurde von der GlassFish Community entwickelt und verbessert.

Apache CXF

Apache CXF ist eine Implementierung von *JAX-WS*, der Java API for XML Web Services und wurde von der Apache Foundation entwickelt und weiterentwickelt.

WildFly besitzt wie die meisten Java EE Anwendungsserver auch individuelle Deployment Deskriptoren: [Redf]

- `jboss-deployment-structure.xml`

  Mit `jboss-deployment-structure.xml` kann das Deployment einer Anwendung beeinflusst werden. Hierzu gehört beispielsweise das Hinzufügen von zusätzlichen Abhängigkeiten, das Definieren von zusätzlichen Modulen oder das Ändern des Ladeverhaltens von Klassen. Das Wurzelelement der zugehörigen XML Struktur heißt `<jboss-deployment-structure>`.

- `jboss-ejb3.xml`

  `jboss-ejb3.xml` ist eine Erweiterung oder Ersetzung des Deployment Deskriptors `ejb-jar.xml` für EJBs ab Version 3. Das äußerste Element im XML Schema hat den Namen `<jboss:ejb-jar>`.

- `jboss-all.xml`

  Der Deployment Deskriptor `jboss-all.xml` erweitert die `application.xml` um zusätzliche Einstellungen, die beispielsweise JPA betreffen können oder EJB, aber für alle Module in der Enterprise Anwendung gelten. Das Wurzelelement des zugehörigen XML Schemas besitzt den Namen `<jboss>`.

- `jboss-app.xml`

  Mit `jboss-app.xml` kann die `application.xml` eines `ear`-Pakets erweitert werden. Das Wurzelelement im XML Schema heißt `<jboss-app>`.

- `jboss-web.xml`

  Eine `web.xml` kann mit dem Deployment Deskriptor `jboss-web.xml` erweitert werden. Das Wurzelelement im zugehörigen XML Schema hat den Namen `<jboss-web>`.

- `jboss-client.xml`

  Mit `jboss-client.xml` kann der Deployment Deskriptor `application-client.xml` eines Anwendungsclient Moduls erweitert werden. Das äußerste XML Element im Schema hat den Namen `<jboss-client>`.

- `jboss-ejb-client.xml`

  Der Deployment Deskriptor `jboss-ejb-client.xml` wird verwendet, um eine Anwendung für den Fernzugriff auf EJBs zu konfigurieren. Das Wurzelelement des XML Schemas heißt `<jboss-ejb-client>`.

- `jbosscmp-jdbc.xml`

  Mit dem Deployment Deskriptor `jbosscmp-jdbc.xml` werden CMP Entitäten auf Datenbankobjekte abgebildet. Die Datei befindet sich im Ordner `META-INF` eines EJB Moduls. Das äußerste Element im Deployment Deskriptor XML Schema heißt `<jbosscmp-jdbc>`.

- `ironjacamar.xml`

  Der Deployment Descriptor `ironjacamar.xml` erweitert den normalen `ra.xml` Deskriptor eines Ressourcen Adapter Moduls. Im XML Schema ist das Wurzelelement mit `<ironjacamar>` benannt.

- `jboss-webservices.xml`

  Mit dem Deployment Descriptor `jboss-webservices.xml` können JBoss Webservices und deren Endpunkte konfiguriert werden. Er befindet sich im Ordner `META-INF` für EJB Webservices oder `WEB-INF` bei Webservices in Webanwendungen. Das Wurzelelement im XML Schemar heißt `<webservices>`.

- `«NAME»-jms.xml`

  Mit dem Deployment Deskriptor `«NAME»-jms.xml` können JMS Nachrichtenempfänger konfiguriert und installiert werden. Der Deskriptor muss entweder im Ordner `META-INF` oder `WEB-INF` liegen, je nach dem welche Art von Anwendung damit konfiguriert wird. `«NAME»` kann durch einen beliebigen Namen ersetzt werden und es kann mehrere dieser Dateien in einem Modul geben. Das äußerste Element im XML Schema hat die Benennung `<messaging-deployment>`.

- `«NAME»-dx.xml`

  Mit dem Deployment Deskriptor `«NAME»-ds.xml` können Datenquellen wie Datenbanken in einem Modul definiert und konfiguriert werden. `«NAME»` kann durch einen beliebigen Namen ersetzt werden und es kann mehrere dieser Dateien in einem Modul geben. Der Deskriptor muss sich entweder im Verzeichnis `META-INF` oder `WEB-INf` befinden. Das Wurzelelement im XML Schema heißt `<datasources>`.

2.3.5 Apache Tomcat

Der *Apache Tomcat* Server ist ein open-source, auf Java basierender Java EE Anwendungsserver, der Servlets und JSPs unterstützt. Früher war Tomcat ein Teilprojekt von Apache Jakarta, wobei er mittlerweile, aufgrund seiner Popularität, ein eigenes Apache Projekt besitzt. Der Apache Tomcat Server unterstützt im Gegensatz zu allen anderen hier vorgestellten Applikationsservern nicht das Full Profile von Java EE, sondern nur das Web Profile, was allerdings für die meisten Anwendungen ausreichend ist. Die aktuelle Version 8 implementiert das Web Profile von Java EE 6. [VG11]

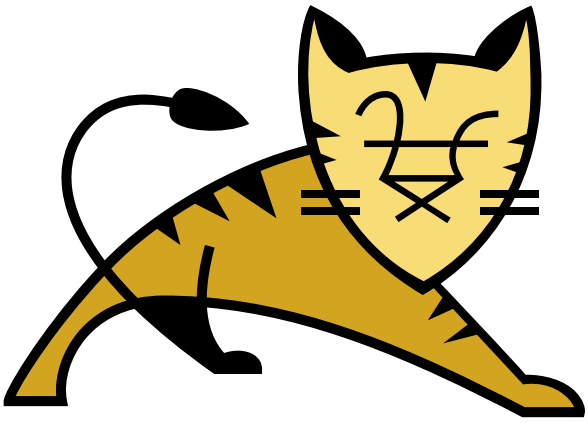


Abbildung 2.10: Das Logo von Apache Tomcat aus [Apab].

Das Logo von Apache Tomcat ist in Abbildung 2.10 zu sehen. Die Hauptkomponente von Tomcat ist *Catalina*, eine Servlet Engine, die in der aktuellen Version die Java Servlet API

in Version 3.0 implementiert. [VG11]
Neben dem normalen Apache Tomcat existiert noch die Erweiterung *Apache TomEE*, die auf Tomcat aufsetzt und mit Java EE Komponenten erweitert, die im normalen Funktionsumfang nicht vorhanden sind. Die Kernkomponenten von TomEE sind: [Apad]

Apache OpenWebBeans

Apache OpenWebBeans ist eine Implementierung von CDI, der Contexts and Dependency Injection API des Java EE Standards. Neben TomEE ist Apache OpenWebBeans auch Teil von IBM WebSphere und IBM WebSphere Liberty Profile in Version 8.5. [Apaa]

Apache OpenEJB

Mit *Apache OpenEJB* besitzt Apache TomEE eine EJB Implementierung, die gleichzeitig auch JAX-RS, JAX-WS, JMS implementiert. Apache OpenEJB ist eigentlich ein eigenständiger EJB Server, aus dem Apache TomEE hervorgegangen ist. [Apad]

Apache OpenJPA

Die JPA Implementierung von Apache TomEE ist *Apache OpenJPA*.

Apache MyFaces

JSF wird von Apache TomEE durch *Apache MyFaces* implementiert.

Daneben gibt es noch *Apache TomEE+* mit den zusätzlichen Erweiterungen: [Apad]

Apache ActiveMQ

Apache ActiveMQ ist die Implementierung von Apache TomEE+ von JMS.

Apache Geronimo Connector

Die Implementierung von JCA in Apache TomEE+ ist *Apache Geronimo Connector*.

Damit ist Apache TomEE+ ein vollwertiger Java EE Anwendungsserver. Apache TomEE implementiert fast alle APIs aus dem Java EE Full Profile, aber das komplette Web Profile und Apache Tomcat implementiert das Java EE Web Profile. Wegen der Standardkonformität von Apache Tomcat ist dieser Server sehr beliebt. Es kommt noch hinzu, dass der Server kostenlos benutzbar ist. Zusätzlich benötigen die wenigsten Java EE Anwendungen Komponenten, die nur im Full Profile des Standards enthalten sind. [VG11]
Apache Tomcat kennt den folgenden individuellen Deployment Deskriptor: [Apab]

- `context.xml`

  Mit dem Deployment Deskriptor `context.xml`, der sich im Verzeichnis `META-INF` einer Java EE Anwendung befindet, können andere Deployment Deskriptoren wie `web.xml` erweitert und teilweise ersetzt werden. Mit diesem Deployment Deskriptor können Einstellungen des Kontexts vorgenommen werden, wie beispielsweise Umgebungsvariablen oder Ressourcen. Das Wurzelelement der `context.xml` heißt `<context>`.

2.4 Ontologie

In den Kapiteln 2.1 bis 2.3 wurden die Grundlagen von Java EE dargelegt und verschiedene Java EE Anwendungsserver beschrieben. In diesem Kapitel wird das Konzept der Ontologie aus Sicht der Informatik vorgestellt und anschließend deren Repräsentationen erläutert. Für diese Arbeit wird eine Java EE Anwendung mit einer Ontologie dargestellt, die Rückschlüsse auf die Migrierbarkeit der Anwendung zulässt.

Nach [SS09] hat das Wort Ontologie zwei Bedeutungen. Zum Einen ist es die philosophische Disziplin, die sich mit der Natur und Struktur der Realität befasst. Aristoteles definierte diese Ontologie als die Wissenschaft des „Seins wegen dem Sein“, also das Studium von Attributen, die aufgrund ihrer Natur zu Dingen gehören.

Die andere Bedeutung von Ontologie ist eine spezielle Art eines Informationsobjekts oder „computational artifact“ in der Informatik. Rechnerische Ontologien sind ein Mittel um die Struktur eines Systems formal zu beschreiben. Damit sind die Elemente und Verknüpfungen gemeint, die aus der Betrachtung des Systems folgen und sinnvoll in Bezug auf das definierte Ziel sind. [SS09]

2.4.1 Beschreibung

Gruber hat 1993 eine Ontologie als „explicit specification of a conceptualization“ – eine explizite Spezifizierung einer Konzeptualisierung definiert [Gru93]. Der Körper von formell beschriebenem Wissen basiert auf einer Konzeptualisierung: Die Objekte, Konzepte, andere Bestandteile, die in einem solchen Interessengebiet existieren, und die Verknüpfungen zwischen all diesen Elementen. Eine Konzeptualisierung ist eine abstrakte, vereinfachte Sicht auf die Welt, die zu einem bestimmten Zweck abgebildet werden soll. Jede Wissensdatenbank, jedes wissensbasierte System und jeder wissensbasierte Agent bezieht sich explizit oder implizit auf irgendeine Konzeptualisierung. [SS09]

Für unser Verständnis reicht es eine Ontologie als abstraktes Modell eines Aspekts der Welt zu betrachten, das explizit spezifiziert ist. Die Ontologie definiert die Eigenschaften aller wichtigen Konzepte und Verknüpfungen. Eine explizite Spezifikation heißt, dass das Modell in einer eindeutigen Sprache beschrieben werden soll, die sowohl von Menschen als auch von Maschinen gelesen, verstanden und weiter entwickelt werden kann. [SS09]

Die wesentliche Grundlage einer Ontologie bildet eine Taxonomie, also eine Klassenhierarchie. Zusätzlich zu dieser Hierarchie gibt es in einer Ontologie noch weitere Verknüpfungen und Relationen zwischen Elementen, Instanzen, Attributen und allen anderen Arten von Objekten. Diese Verknüpfungen können beispielsweise selbst Attribute darstellen oder Beziehungen zwischen einzelnen Objekten widerspiegeln. Zusätzlich können in einer Ontologie auch Regeln und Bedingungen definiert werden, die anhand der vorhandenen Daten neue Informationen liefern können. [NM01; Stu11; AA12; UJ99]

2.4.2 Modellierungs- und Präsentationssprachen

In diesem Kapitel wird kurz auf maschinenlesbare Sprachen und Formate eingegangen, mit denen eine Ontologie modelliert oder präsentiert werden kann. Zu erwähnen sind hier XML, RDF und OWL.

2.4.2.1 Extensible Markup Language – XML

Die *Extensible Markup Language* (XML) ist eine Sprache, um hierarchisch strukturierte Daten darzustellen. Strukturelle und inhaltliche Einschränkungen und Definitionen können über die Verwendung von *XML Schema Definitionen* (kurz *XSD*) vorgenommen werden. Dieses Schema wird in einer `xsd`-Datei beschrieben, die wieder einem XML Schema folgt. Eingebunden werden solche XML Schemata über das Attribut `xmlns` im Wurzelelement der XML Datei. Eine ältere Methode, um eine XML Struktur zu definieren, geht über *Document Type Definition* Dateien. Mittlerweile wird aber von *World Wide Web Consortium* vorgeschlagen, ein XML Schema mittels XSD festzulegen. [BYM+08]
XML ist für das Darstellen von Ontologien aufgrund seiner hierarchischen Struktur geeignet. Grundsätzlich kann alles mit XML repräsentiert werden. Der Hauptaufwand hierbei ist das Definieren des verwendeten Schemas, damit die Informationen auch wirklich von einem Programm ausgelesen und verstanden werden können.
Im Prinzip sind alle hier vorgestellten Sprachen Erweiterungen von XML mit einem für Ontologien angepassten Schema.

2.4.2.2 Resource Description Framework – RDF

Das *Resource Description Framework* ist ein Framework, um Informationen im Web zu beschreiben, kann aber prinzipiell auch dafür verwendet werden beliebige Ressourcen logisch zu definieren. Die Kernstruktur von RDF ist eine Menge von *Tripeln*, wobei jedes ein Subjekt, ein Objekt und ein Prädikat enthält. Jedes dieser Tripel wird *RDF Graph* genannt. In Abbildung 2.11 ist ein solches Tripel einfach dargestellt. Abgesehen von der graphischen Darstellung wird RDF als Erweiterung von XML gesehen und entsprechend hierarchisch mit eigenen Tags geschrieben. [KCM14]

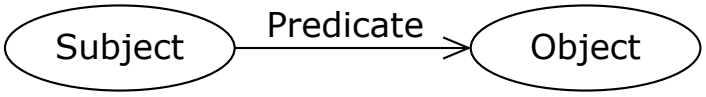


Abbildung 2.11: Beispiel eines RDF Graphen mit zwei Knoten (Subject und Object) und einem *Tripel*, das beide verbindet (Predicate), aus [KCM14, Fig. 1].

Jeder Knoten repräsentiert eine Klasse oder Instanz von beschriebenen Ressourcen. Ein RDF Tripel legt fest, dass eine Verbindung, identifiziert durch das Prädikat, zwischen den Ressourcen Subjekt und Objekt vorhanden ist. Diese Aussage wird auch *RDF Statement* bezeichnet. Das Prädikat beschreibt also eine Eigenschaft zwischen genau zwei Ressourcen. Relationen zwischen mehr als zwei Entitäten können in RDF nur indirekt dargestellt werden. [KCM14]
Eine semantische Erweiterung des normalen RDF bietet *RDF Schema* (RDF-S). Im Gegensatz zu gängigen objektorientierten Sprachen werden Klassen in RDF-S nicht anhand ihrer Eigenschaften beschrieben, sondern Eigenschaften hinsichtlich ihrer zugehörigen Klassen. Auf diese Weise können Erweiterungen am kompletten System vorgenommen werden ohne die grundlegende Struktur zu verändern. [GS14; GB14]
Mit RDF und speziell RDF-S ist eine Erweiterung von XML gegeben, in der die grundlegende Struktur einer Ontologie sehr gut abgebildet werden kann. Gegenüber normalem XML hat RDF den Vorteil, dass kein Schema definiert werden muss, weil die entsprechenden Objekttypen wie Klassen und Relationen schon vorhanden sind. Weiterhin ist es hier nicht

mit vorhandenen Mitteln möglich Regeln oder Bedingungen für die Ontologie zu definieren. RDF und RDF-S bieten sich als Präsentationssprache für eine graphische Darstellung einer Ontologie an.

2.4.2.3 Web Ontology Language – OWL

Die *Web Ontology Language* (OWL) ist eine auf XML und RDF basierende Sprache zur Repräsentation von Ontologien. Hierbei wird der Fokus auf die applikationsgestützte Verarbeitung der vorhandenen Informationen gelegt, im Gegensatz zur graphischen Präsentation für Menschen. OWL kann dafür benutzt werden, die Bedeutung von Objekten und den Relationen zwischen diesen zu definieren. Mit OWL können Bedeutung und Semantik einer Ontologie einfacher beschrieben werden als mit XML, RDF oder RDF-S. Durch diese Erweiterungen im Vokabular gegenüber den vorgestellten anderen Sprachen, können Inhalte leichter von Maschinen gelesen und interpretiert werden. [HM04]
OWL tritt in den folgenden drei Varianten auf, die sich hauptsächlich im Grad der Expressivität unterscheiden: [HM04]

- **OWL Lite**

  OWL Lite unterstützt einfache Hierarchien und Relationen und hat die niedrigste Expressivität der OWL Sprachen. Beispielsweise kann die Kardinalität einer Relation oder Eigenschaft nicht höher als zwei sein. Im Grunde ist OWL Lite die Form von OWL, in der normale Taxonomien dargestellt werden können. Richtige Ontologien mit Regeln, Bedingungen und höheren Kardinalitäten sind nicht gestattet.

- **OWL DL**

  OWL DL unterstützt die maximale Expressivität und garantiert die rechnerische Vollständigkeit und Entscheidbarkeit. Das bedeutet, dass alle Schlussfolgerungen garantiert berechnet und in endlicher Zeit gezogen werden können. Das DL in OWL DL steht für *description logics*, einem Forschungsbereich, der sich mit der Logik befasst, die die formale Grundlage von OWL darstellt. OWL DL beinhaltet alle OWL Sprachkonstrukte, wobei diese nur unter gewissen Voraussetzungen verwendet werden können. Beispielsweise kann eine Klasse zwar Subklasse von mehreren Klassen sein, aber nicht eine Instanz einer anderen Klasse.

- **OWL Full**

  OWL Full unterstützt wie OWL DL die maximale Expressivität und lässt einem Anwender die vollen syntaktischen Freiheiten von RDF ohne rechnerischer Garantien. Vorteil davon ist, dass das vorher definierte Vokabular (RDF oder OWL) erweiterbar ist. Allerdings ist es dann unwahrscheinlich, dass eine Anwendung wirklich alle möglichen Schlussfolgerungen ziehen kann.

2.4.2.4 andere Formate

Neben den hier vorgestellten Formaten gibt es noch viele weitere Formate, mit denen Ontologien dargestellt werden können. Die folgende Liste enthält andere nicht in dieser Arbeit benötigte Sprachen: [UG96; LGJ+98]

- Interchange Format – PIF

- Knowledge Interchange Format – KIF
- Semantic Web Rules Language – SWRL
- Description Logic Programs – DLP
- Frame Logic – F-logic
- $\mathcal{SHIQ}$

Es gibt einige Programme, um Ontologien zu entwickeln und sie in der entsprechenden Sprache abzuspeichern. Hierbei sind vor allem „Protégé“ und SWOOP zu nennen. Es gibt aber beispielsweise auch ein Eclipse Plugin, mit dem Ontologien erstellt werden können.

3 Themenverwandte Arbeiten

In diesem Kapitel werden exemplarisch einige Arbeiten vorgestellt, die sich mit der Migration von Applikationen und Migrationsontologien befassen. Die Beispiele in den Kapiteln 3.1 und 3.2 stellen einen Überblick über die vorhandenen Hilfsmittel im Internet dar, um Applikationen zwischen spezifischen Applikationsservern zu migrieren. In Kapitel 3.3 werden Ontologien vorgestellt, die eine Migration beschreiben sollen oder als Austauschsprache konzipiert sind.

3.1 Migrationsbeschreibungen

In diesem Kapitel werden zwei Migrationsbeschreibungen - für Apache Tomcat und IBM WebSphere - vorgestellt, die die Hersteller von Applikationsservern frei zugänglich zur Verfügung stellen. Diese Beschreibungen sollen Entwickler darin unterstützen ihre Applikation auf den spezifischen Server umzuziehen. Zusätzlich werden Vorteile des Applikationsservers herausgearbeitet.

3.1.1 Tomcat Migration Guide

Die Webseite [Apac] ist ein typisches Beispiel einer Migrationsbeschreibung im Internet. Hier werden Anleitungen zu Migration von Anwendungen zwischen unterschiedlichen Versionen von Apache Tomcat gegeben. Da Apache Tomcat nicht den kompletten Java EE Standard unterstützt existieren keine offiziellen Migrationsanleitungen zwischen anderen Java EE Servern von Apache.
Die folgenden Migrationsszenarien werden auf [Apac] einzeln aufgeführt:

- Migration von den Tomcat Versionen 5.5.x zu 6.0.x
- Upgrade zwischen Tomcat Versionen 6.0.x
- Migration von den Tomcat Versionen 6.0.x zu 7.0.x
- Upgrade zwischen Tomcat Versionen 7.0.x
- Migration von den Tomcat Versionen 7.0.x zu 8.0.x
- Upgrade zwischen Tomcat Versionen 8.0.x

Jede einzelne Migrationsanleitung ist gegliedert nach Änderungen zwischen den behandelten Versionen wie beispielsweise verwendete Java Version, unterschiedliche Ordner Struktur oder andere Versionen verwendeter APIs. Zu jeder dieser Änderungen sind Beispiele aufgeführt, wie vorhandene Elemente geändert werden müssen, damit der neuere Server sie verstehen und interpretieren kann. Zusätzlich zu diesen Code betreffenden Unterschieden wird noch auf Unterschiede in Konfigurationsdateien eingegangen. Hierfür können die Standardkonfigurationsdateien jeder einzelnen Tomcat-Version untereinander verglichen werden. [Apac]

3.1.2 WebSphere Application Server Migration Guides

Mit dem von IBM veröffentlichten Dokument „WebSphere Application Server V8.5 Migration Guide“ [ACE+12] sollen Migration zu IBM WebSphere V8.5 deutlich vereinfacht werden. Neben dem Migrationsanleitung für Version 8.5 existieren auch Dokumente für die Versionen 5 [YSD+03], 6 [CAB+06] und 7 [CDK+10], wobei hier nur auf Migrationen von früheren Versionen von WebSphere auf die explizite Version eingegangen wird.
Aufgebaut sind alle diese Migrationsanleitungen nach dem gleichen Schema: Als erstes werden die Konzepte und Architektur des WebSphere Servers beschrieben. Anschließend werden Migrationsmethodik und Gründe für eine Migration angesprochen. Zuletzt wird auf die Migration von Applikationen verschiedener Server eingegangen.
Interessant ist neben den technischen Möglichkeiten des WebSphere Servers und der Beschreibung der Migration an sich vor allem der Teil über die Migrationsmethodik. Hier werden Methodik und Migrationsziele genannt, die unabhängig vom Servertyp sind, da sie auf alle gleichermaßen zutreffen.
Nach [ACE+12] besteht eine Migration aus den folgenden Teilen:

- **Migrationsbewertung**

  Die Migrationsbewertung ist der Beginn einer jeden Migration. Hier werden Stakeholder nach ihren Wünschen befragt und eine Bestandsaufnahme der Ziele, der Ausgangssituation, der beteiligten Personen etc. durchgeführt.

- **Migrationsplanung**

  In der Migrationsplanung wird ein Projektplan zur Migration erstellt. Zu den wichtigsten Bestandteilen und Grundlagen eines Projektplans gehören Rollen, Verantwortlichkeiten, Testpläne, Aufgaben, Deadlines, Risiken und Dokumentation.

- **Migrationsimplementierung**

  Die tatsächliche Migration findet in der Migrationsimplementierung statt. Hierzu gehören nicht nur die Umsetzung der Migration, sondern auch das Vorbereiten, das Testen, der Rollout der migrierten Anwendung und das anschließende Aufräumen (siehe Abbildung 3.1).

- **Migrationsnachbearbeitung**

  Mit diesem Teil des Migrationsprozesses ist das Verbessern und Optimieren der Anwendung nach der eigentlichen Migration gemeint. Das heißt die Applikation läuft bereits auf dem neuen Server, soll aber noch dahingehend optimiert werden, dass sie auch schneller oder effizienter läuft.

Obwohl Java EE eine Framework definiert, das unabhängig vom installierten Anbieter ist, führen einzigartige Implementierungen der Java EE Spezifikation zu Problemen bei der Migration von Java EE Applikationen. Anbieter implementieren zusätzliche Funktionen, die nicht im Java EE Standard enthalten sind. Das ist der Grund, warum Oracle eine Liste von Anbietern mit Java EE kompatiblen Produkten anbietet. Theoretisch sollten Anwendungen, die auf einem Java EE Applikationsserver deployt sind, auch auf einem anderen Server laufen. Hierbei wird beispielsweise auf Kompatibilität zwischen unterschiedlichen Java EE Applikationsservern und Applikationsportierbarkeit eingegangen. [ACE+12]
Genaue Migrationsbeschreibungen zwischen verschiedenen Servern und nicht nur Versionen sind vorhanden für [ACE+12]:

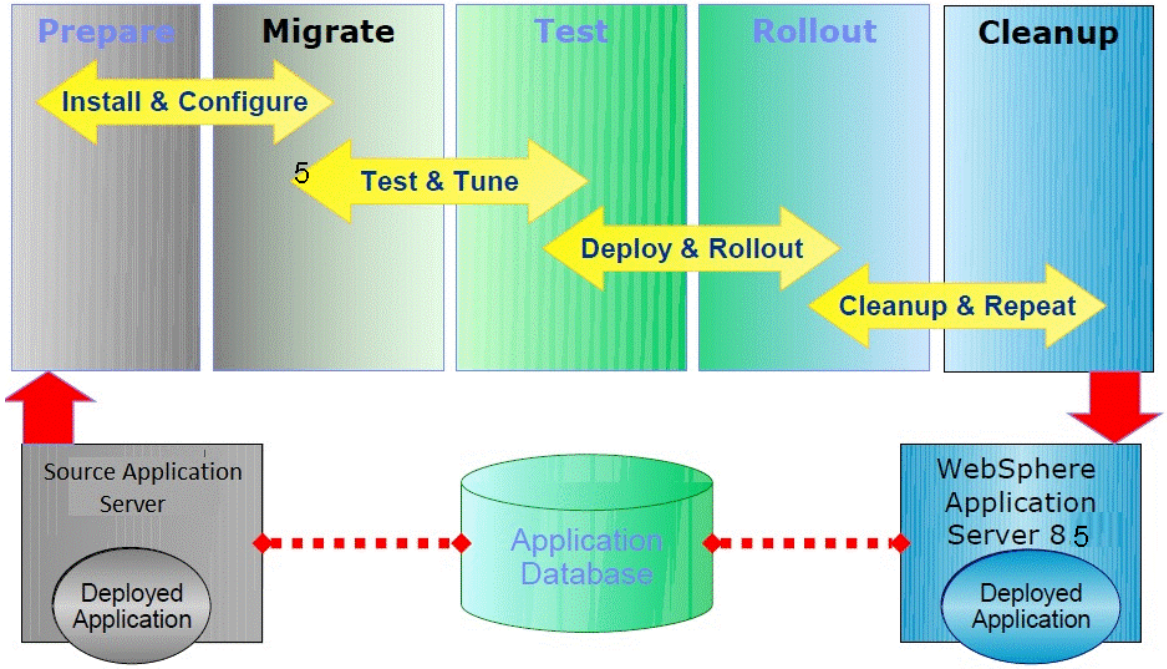


Abbildung 3.1: Migrationshauptaufgaben aus [ACE+12]

- Oracle Weblogic 10
- Oracle Application Server 10
- JBoss 7
- Apache Tomcat 7

Bei Migrationen von älteren Versionen von WebSphere wird auf das in Abschnitt 3.2.2 beschriebene „IBM Websphere Application Server Migration Toolkit“ verwiesen. Zusätzlich kann mit diesem Toolkit auch von anderen Applikationsservern migriert werden, wie beispielsweise Apache Tomcat, JBoss Application Server, Oracle Application Server oder Oracle WebLogic Server. [IBMb]

3.2 Migrationstools

Neben den in Kapitel 3.1 beschriebenen Migrationsanleitungen existieren auch einige Tools, die Teile einer Migration automatisieren oder zumindest Hilfestellungen geben können. In diesem Kapitel wird beispielhaft das Glassfish-Upgrade Tool, das WebSphere Application Server Migration Toolkit und Windup betrachtet.

3.2.1 Glassfish - Upgrade Tool

Das von Oracle mit GlassFish 3.1.2 ausgelieferte Upgrade Tool ermöglicht eine Applikationsmigration von einer älteren GlassFish Version zur Version 3.1.2. Das Upgrade Tool unternimmt hierbei keine Änderungen an der Anwendung selbst, sondern passt nur die Konfiguration des Servers anhand der alten Konfiguration an. Das bedeutet auch, dass eine

funktionierende Version der Anwendung mit altem Server vorhanden sein muss, aus der das Upgrade Tool die Informationen beziehen kann. Anwendungen müssen nicht verändert werden, da sie beispielsweise mit der Einstellung `compatibility = v2` gestartet werden können, um die gleiche Funktionalität wie in Version 2 von GlassFish zu erhalten. [Oraf]

Damit ist das GlassFish Upgrade Tool kein wirkliches Migrationstool, sondern wie der Name schon sagt ein Upgradetool. Es nimmt keine Veränderungen an der Applikation vor, passt jedoch die Servereinstellungen den alten Einstellungen an. Das ist für eine Migration auch essentiell wichtig, stellt aber nicht sicher, dass eine Applikation auch technisch auf dem Server lauffähig ist.

3.2.2 WebSphere Application Server Migration Toolkit

Das „IBM WebSphere Application Server Migration Toolkit“ ist eine Sammlung von Tools und Wissensdatenbanken, die bei einer Migration zu WebSphere in den Versionen 7.0 bis 8.5.5 unterstützen soll. Dabei ist es nicht von Bedeutung, ob dabei von einer früheren Version von WebSphere oder einem konkurrierenden Applikationsserver wie beispielsweise Apache Tomcat, JBoss Application Server, Oracle Application Server oder Oracle WebLogic Server migriert werden soll. [IBMb]
Zu diesem Toolkit gehören die in Abschnitt 3.1.2 vorgestellten Migrationsanleitungen sowie die folgenden Tools [IBMb]:

- Apache Tomcat to Liberty Profile Configuration Migration Tool
- Apache Tomcat to WebSphere Application Migration Tool
- JBoss to WebSphere Application Migration Tool
- Oracle to WebSphere Application Migration Tool
- WebLogic to WebSphere Application Migration Tool

Jedes dieser Migrationstools kann in IDEs wie Eclipse eingebunden werden und erkennt Komponenten, die aktualisiert werden sollten, um optimale Kompatibilität und Leistung zu gewährleisten. Es werden unter Anderem die folgenden Aspekte einer Applikation betrachtet [IBMb]:

- Proprietäre Deployment Deskriptorkonfigurationen von Java EE Komponenten
- Proprietäre Java API Paketreferenzen
- Proprietäre Java Annotations
- Nicht portierbare JNDI Suchstrings und InitialContext Konfiguration
- Nicht standardkonforme JSP-Verwendung und -Konstrukte

Zusätzlich existiert seit März 2015 für IBM WebSphere Liberty Profile das „Migration Toolkit for Application Binaries (Tech Preview)“. Nach [IBMa] ist IBM WebSphere Liberty Profile ein leichtgewichtiges WebSphere Profil mit einfacherer Konfiguration und einigen Open

Source Komponenten. Dieses Toolkit analysiert eine Applikation und erzeugt einen detaillierten Report mit Hilfestellungen und potentiellen Lösungen für problematische Aspekte der Anwendung. [Kin]

Damit sind die hier erwähnten Migrationstools eine sehr umfangreiche Möglichkeit um auf die WebSphere Versionen 7.0 bis 8.5.5 zu migrieren. Es gibt allerdings einige nicht betrachtete und dennoch häufig verwendete Servertypen wie beispielsweise Oracle GlassFish oder Jetty.

3.2.3 JBoss - Windup Migration Platform

Die Windup Migration Platform, kurz Windup, ist ein von Red Hat entwickeltes und unter der Eclipse Public License v.1.0 stehendes Open Source Tool, mit dem Ziel eine Java EE Anwendung zu analysieren und anschließend einen Report bezüglich der Migrierbarkeit zu JBoss auszugeben [Dav]. Prinzipiell kann jeder an dem Projekt mitarbeiten und es erweitern. Dazu ist es nicht nötig selbst zu programmieren, da auch Erweiterungen in der XML-Konfiguration der Analyse- und Reportregeln Verbesserungen bringen können. [Reda]
Der ausgegebene Report wird in Form einer html-Seite erstellt und klassifiziert die gefundenen Probleme anhand ihres Migrationsaufwandes. Es werden neben den Deploydeskriptoren auch die Klassen und Struktur der Java EE Anwendung analysiert. Durch Markieren der entsprechenden Stellen und Vorschläge soll die Migration erleichtert werden. Zusätzlich kann Windup auch einige Klassen und Konfigurationen selbstständig für JBoss umwandeln, was weiteren Aufwand sparen kann. Ein Eindruck über den von Windup erstellten Report kann aus Abbildung 3.2 gewonnen werden. [Redb]

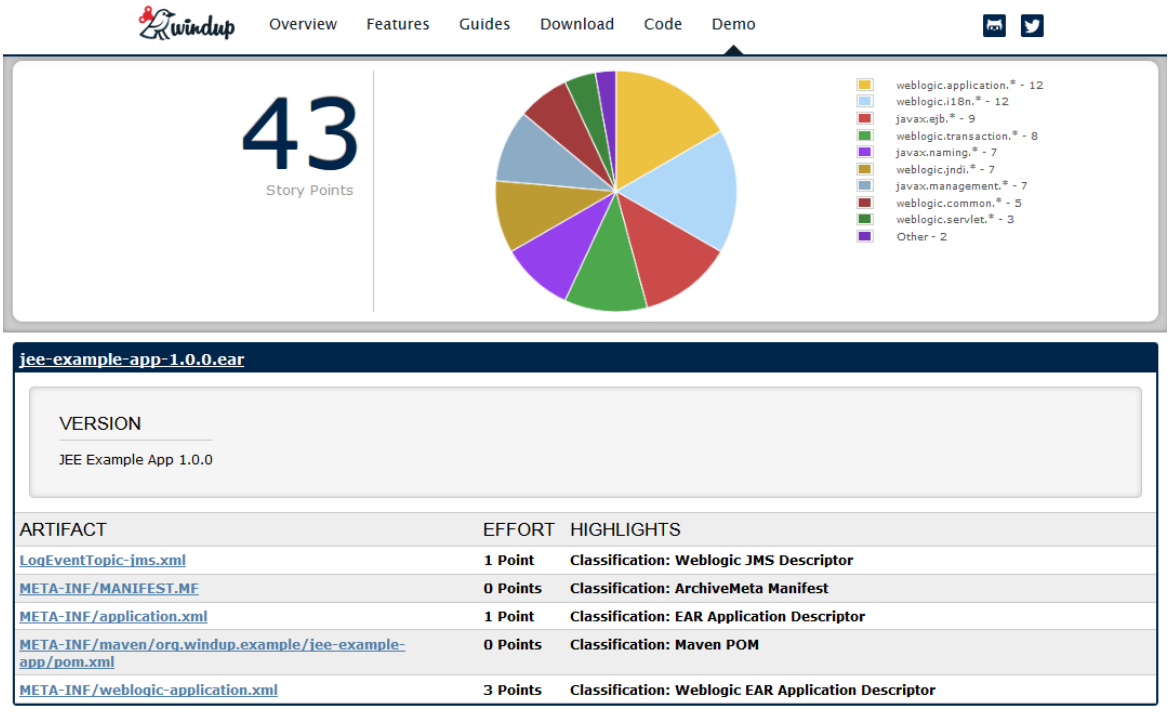


Abbildung 3.2: Ausschnitt aus dem Windup Demo Report (von [Redb, example.html]).

Windup ist eine Standalone Anwendung, die auch nicht in IDEs wie Eclipse integriert werden

muss oder kann. Durch die Möglichkeiten zur Anpassung und Erweiterung der Regeln ist es möglich mit Windup jede Art der Migration zu unterstützen. Der eigentliche Fokus von Windup liegt jedoch in der Migration von Java EE Anwendungen von einem beliebigen Applikationsserver zu JBoss. Windup besitzt die folgenden drei Analysatoren, Interrogatoren genannt: [Reda]

- **Java Interrogator**

  Dieser Analysator liest kompilierte Java Dateien und erkennt ob Importe auf der internen Blackliste stehen. Falls das der Fall ist, wird die Klasse weiter analysiert.

- **JSP Interrogator**

  Dieser Analysator bearbeitet JSP Dateien anhand ihrer Importe.

- **XML Interrogator**

  Dieser Analysator liest XML-Dateien in ein DOM Objekt ein und erkennt nicht verfügbare Konfigurationen.

Windup stellt mit der Erweiterbarkeit und einfachen Konfiguration eine wirklich gute Basis für Migrationen bereit, besitzt allerdings aktuell nur eine Unterstützung für JBoss und keinen anderen Applikationsserver, obwohl das prinzipiell möglich wäre. Nichtsdestotrotz ist das anschauliche Ergebnis ein sehr gutes Beispiel dafür, was Migrationstools aktuell leisten können.

3.3 Ontologien

In den vorangegangenen Kapiteln wurden Anleitungen und Hilfsmittel zur Migration von Java EE Anwendungen beschrieben. Keines dieser Hilfsmittel verwendet eine Ontologie, um Daten zu speichern oder darzustellen. Der Vorteil einer Ontologie ist es, dass auf Repräsentationsebene Zusammenhänge zwischen einzelnen Objekten dargestellt werden können, aus denen einfach Schlussfolgerungen über definierte Bedingungen gezogen werden können.
In diesem Kapitel werden beispielhaft einige Ontologien vorgestellt, die als Hilfsmittel in einer Migration oder als Zwischensprache verwendet werden. Eine Kombination aus Zwischensprache und Migrations-Ontologie ist in der Literatur nicht bekannt. Die hier erwähnten Beispiele werden Einfluss auf die Konstruktion und die Bestandteile der später in Kapitel 5 entwickelten Ontologie haben.
Da Computer immer allgegenwärtiger werden und die Informationen, die sie sammeln und speichern immer mehr werden, nimmt der Nutzen von formell repräsentierten Informationen weiter zu. Ontologien sind eine der Haupttechnologien, um Informationen eine klar definierte Bedeutung zu geben. Das erlaubt Computern und Menschen kooperativer und damit effizienter zu arbeiten. [SMMS02]

3.3.1 Datenmigration zwischen Datenbanken

In dem Artikel „Ontology based Secured Data Migration“ wird auf die Migration von Daten zwischen verschiedenen Datenbanken eingegangen, wobei ein Ontologie für die Migration verwendet wird. Datenmigration ist der Prozess Daten zwischen verschiedenen Speichertypen,

Formaten oder Computersystemen zu transferieren. Da die Menge der Daten eine manuelle Migration in den meisten Fällen ausschließen, soll der Prozess automatisiert von Statten gehen. Zusätzlich wird ein flexibles Transformationsmodell benötigt, sodass der Migrationsprozess vielfältig angewendet werden kann. [SVS12]
Die hier beschriebene Ontologie besteht aus zwei Teilontologien, nämlich der lokalen Ontologie mit der Struktur der alten Datenbank und der globalen Ontologie mit der Struktur der neuen Datenbank. Die Repräsentation der Ontologie wird mit XML realisiert. Als erstes wird die Struktur der alten Datenbank auf die lokale Ontologie abgebildet und analysiert. Anschließend wird diese Struktur auf die globale Ontologie transformiert. Hierbei werden Datenbanktypen und Attribute so in der globalen Ontologie angepasst, dass die Daten der lokalen Ontologie integriert werden können. Anschließend wird versucht eine passende Transformation der Daten der alten Datenbank auf das eventuell veränderte Schema in der globalen Ontologie zu finden. Ist dieser Schritt erfolgreich, so werden die Änderungen an der globalen Ontologie auf die neue Datenbank übertragen und die Daten aus der alten Datenbank transformiert und in der neuen Datenbank gespeichert. Diese Übertragung erfolgt verschlüsselt, um Angreifern nicht die Möglichkeit zu geben, die Daten auszulesen oder zu verändern. [SVS12]
Durch die Verwendung der Ontologie wurde die Migrationsdauer von Daten in Datenbanken deutlich reduziert und eine klare Schnittstelle geschaffen, die Datenmigrationen vereinfacht. Eine vollautomatisierte Implementierung dieses Prozesses muss Schnittstellen zu aktuellen Datenbanksystemen aufweisen, um die Struktur der Datenbank und die Daten darin auslesen zu können. [SVS12]

Es sind keine Implementierungsdetails oder Details der verwendeten Ontologie bekannt. Aus diesem Grund können wenn überhaupt nur Vorgehen bei der Datenbank-Migration für die Migration von Java EE Anwendungen abgeleitet werden. Ob der hier beschriebene Prozess auf unsere Migrationsart anwendbar oder adaptierbar ist, ist aufgrund der Komplexität des Prozesses fraglich.

3.3.2 Servermigration

Die Servermigration stellt eine besondere Art der Datenmigration dar und besteht aus den Teilen Speicherplatzmigration, Datenbankmigration, Applikationsmigration und Geschäftsprozessmigration. Aus diesem Grund ist die Servermigration eine Erweiterung der in Abschnitt 3.3.1 vorgestellten Datenmigration zwischen Datenbanken. Der Prozess der Servermigration übersetzt ebenso wie die Datenmigration Daten von einem Format in ein anderes und ist notwendig, wenn eine Organisation sich für die Benutzung neuer Computersysteme entscheidet. Hierbei minimiert ein automatisierter Prozess die Notwendigkeit von menschlichen Eingriffen und die Ausfallzeit des Systems. [KKS14]
Wie in Abbildung 3.3 gezeigt besteht die Migration aus den Teilen [KKS14]:

- **Query**

  Abfragen der Struktur und Daten im Altsystem.

- **Validate**

  Validierung ob mit diesen Informationen eine Migration möglich ist.

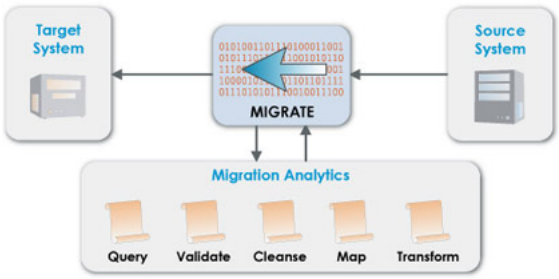


Abbildung 3.3: Datenmigrationsarchitektur aus [KKS14, Fig 1].

- **Cleanse**

  Eventuell nicht benötigte Teils entfernen und Daten aufräumen und sortieren.

- **Map**

  Eine Abbildung definieren, mit der die Daten von der Struktur des alten auf die des neuen Servers übertragen werden können.

- **Transform**

  Die Daten mit der im *Map*-Schritt definierte Abbildung transformieren.

Insbesondere besteht die Servermigration auch aus der Migration der Anwendung selbst, also aus der Überführung einer Applikation von einer Umgebung in eine neue. Hierbei sollte unterschieden werden zwischen der reinen Umgebungsänderung (beispielsweise unterschiedliche Betriebssystemversionen) und der Applikationsserveränderung (beispielsweise von WebLogic nach JBoss).
Beim ersten Fall muss auf der neuen Umgebung ein gleicher Applikationsserver installiert werden, worauf dann die Anwendung deployed wird. Hier kann es beispielsweise dann zu Problemen kommen, wenn der Applikationsserver das neue Betriebssystem nicht unterstützt. In einer solchen Situation muss auch der Applikationsserver verändert werden, was im zweiten und aufwändigeren Fall mündet, weil die Architektur der Anwendung selbst verändert und eine neue Serverkonfiguration durchgeführt werden muss.

Der hier beschriebene Artikel befasst sich ausschließlich mit der reinen Servermigration ohne Applikationsservermigration. Die automatisierte Migration zwischen verschiedenen Applikationsservern wurde bislang nicht untersucht, da ein übergreifendes Protokoll zur Analyse unterschiedlicher Serverarchitekturen fehlt. Die in dieser Arbeit entwickelte Ontologie soll durch entsprechende Erweiterungen eine Anwendung automatisiert analysieren und auswerten. [KKS14]

3.3.3 Web-Service Migration

Nach [KR13a] müssen service-orientierte Softwaresysteme, die in sehr dynamischen oder mobilen Umgebungen arbeiten, dazu fähig sein zur Laufzeit auf wechselnde Umgebungsbedingungen zu reagieren und sich entsprechend anzupassen. Deshalb ist es für diese Softwaresysteme essenziell, dass die Möglichkeit zur Migration besteht und diese soweit möglich automatisiert von statten gehen kann.

Der Ansatz von Kazzaz und Ryhlý, um diesen Migrationsprozess zu ermöglichen, ist ein von der Implementierung unabhängiges Framework. Durch die Einführung von komplett anpassbaren Migrationsstrategien und die Verwendung der beschriebenen Abstraktion ist dieser Ansatz in vielen verschiedenen Anwendungsbereichen flexibel einsetzbar. Die so definierten Migrationsstrategien bewerten Migrationsbedingungen und treffen Migrationsentscheidungen zur Laufzeit der Web-Services. [KR13a]
Die hier verwendete Ontologie wurde mittels der *Semantic Application Design Language* (SADL) [Cra11] erfasst. Das Basiselement in SADL ist eine *class* (Klasse), wobei jede solche Klasse *properties* (Eigenschaften) besitzt. Eine dieser Eigenschaften kann einen oder mehrere Werte von SADL Datentypen oder vorher definierten Klassen enthalten.
Ein *Service* ist in dieser Ontologie dann migrierbar, wenn alle Bedingungen für die Migration erfüllbar sind und ein *MigratableServiceProfile* besitzt. Diese Bedingungen werden als Regeln hinterlegt und geprüft. Am Ende der Analyse wird eine Migrationsentscheidung getroffen, welcher Service wohin migriert werden kann und wird. Die Analyse der Web-Services und damit die Befüllung der Ontologie mit Werten und Profilen wird hierbei automatisiert vom WS-Discovery Server [NR09] vorgenommen. Dieser WS-Discovery Server erzeugt RDF-Resourcen, die entweder einen *ServiceProvider* oder *Service* repräsentieren. Die resultierenden RDF-Resourcen bilden ein RDF-Modell, das die Eigenschaften, die internen Beziehungen und den Status des aktuellen Systems widerspiegeln. Dieses Modell wird durchgehend vom WS-Discovery Server aktualisiert und mit Informationen versorgt. [KR13b]

Diese Arbeit beschreibt eine komplett automatisierte Migration von Web-Services über eine Ontologie, die permanent durch einen WS-Discovery Server mit Informationen versorgt und aktualisiert wird. Hierbei ist zu beachten, dass die Informationen nur so gut sind, wie der WS-Discovery Server. Da es keine Komponente gibt, die Java EE Anwendungen überwacht und kategorisiert, ist dieser Ansatz leider nicht möglich für die Migration von Java EE Applikationen. Es besteht allerdings die Möglichkeit, Teile der Grundontologie wiederzuverwenden, wie beispielsweise Regeln für die Migrierbarkeit von einzelnen Komponenten.

3.3.4 Kactus

Das *Kactus* Projekt befasste sich mit einem applikationsgetriebenem Ansatz zur Wiederverwendung von Wissen im technischen Umfeld. Ontologien sind hier als unterschiedliche Blickwinkel auf das betrachtete System zu sehen. Das kommt daher, dass die Ontologien unterschiedliche Parameter, Bedingungen, Komponenten und Funktionen beinhalten, die das System unterschiedlich beschreiben. Ontologien können hierbei nicht nur Aspekte des Systems beschreiben, sondern auch zusätzliche Informationen durch Übersetzungen zwischen diesen unterschiedlichen Beschreibungen generieren (siehe Abbildung 3.4). Hiermit wird Problemlösern das Wissen zur Verfügung gestellt, das benötigt wird, um eine richtige Lösung zu finden. *CommonKADS* ist diese Methodik des Umformulierens und Neuinterpretierens von schon vorhandenem Wissen. Sie ist wichtig, um Wissen besser verteilen und wiederverwenden zu können. [SWJ95]
Für diese Methodik existiert mit *CML* (Conceptual Modelling Language) eine strukturierte, halbformale Notation. Sie beinhaltet neben beispielsweise Konzepten, Komponenten, Rollen und Bedingungen auch Übersetzungen. Diese Übersetzungen sind der wichtigste Teil dieser Ontologie und bestehen selbst wiederum aus Konzepten, Relationen zwischen einzelnen Konzepten, Eigenschaften von Konzepten und Standardwerten von Eigenschaften von Konzep-

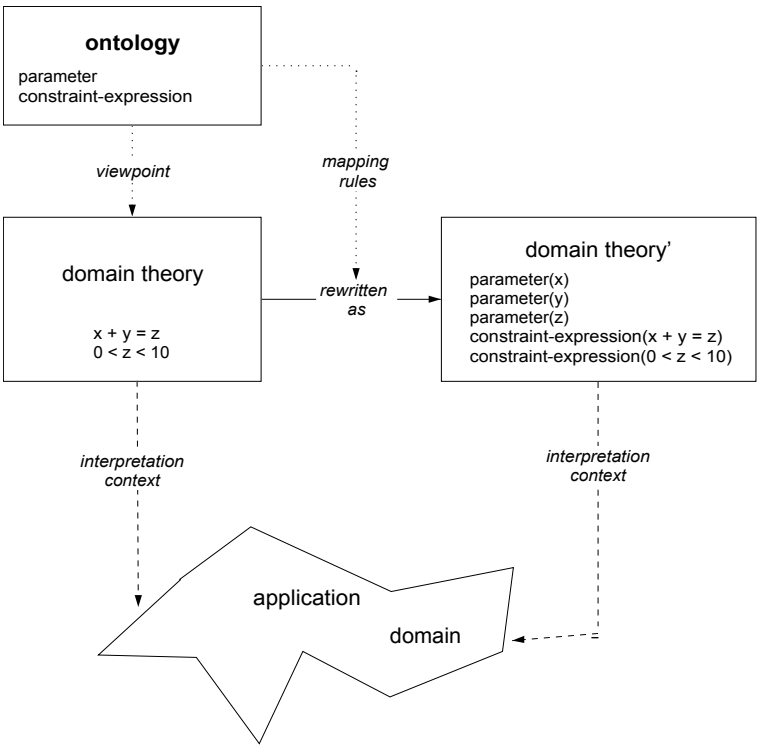


Abbildung 3.4: Ontologie mit Übersetzung eines Systems in der Kactus Methodologie aus [SWJ95, Figure 2].

ten. Mit einer solchen Übersetzung kann eine Repräsentation von Wissen mit entsprechenden Funktionen in eine Ontologie abgebildet werden. [SWA+94]

Der Ansatzpunkt der Übersetzungen zwischen Wissensrepräsentationen und Ontologien spielt für diese Arbeit eine große Rolle. In dem hier vorliegenden Fall ist die analysierte Anwendung das betrachtete System, von dem aus auf eine Ontologie übersetzt werden soll. Dieses Konzept wird in der hier entwickelten Ontologie verwendet werden. Im Gegensatz zur halbformalen CML-Notation besteht der Unterschied in der verwendeten Notation darin, dass sie so formal wie möglich gehalten werden muss.

3.3.5 STEP

STEP – der *Standard for the Exchange of Product Model Data* – ist eine Zwischensprache, um Produkte zu definieren und zu spezifizieren, und ist unter ISO 10303 als Norm hinterlegt. Motivation für die Entwicklung von STEP war es eine Möglichkeit zu finden, um Produktdaten, die auch den kompletten Produktlebenszyklus enthalten, zwischen verschiedenen Computersystemen und Umgebungen auszutauschen. Darin inbegriffen sind unter Anderem Konstruktion, Produktion, Verwendung, Wartung und Entsorgung. Hierbei sind sehr viele unterschiedliche Computersysteme involviert, die auch verschiedenen Organisationen gehören können. Um das zu gewährleisten, müssen Organisationen fähig sein, ihre Produktinformationen in einem gemeinsamen, computerlesbaren Format abzuspeichern. Dieses Format muss vollständig und konsistent bleiben, auch wenn es zwischen verschiedenen Systemen ausge-

tauscht wird. [UG96]
Damit diese Bedingungen erfüllt werden können, stellt STEP einen Mechanismus zur Verfügung, der Produktdaten über den kompletten Lebenszyklus des Produkts unabhängig von einem speziellen System beschreiben kann. Damit ist STEP nicht nur für neutralen Datenaustausch, sondern auch als Grundlage der Datenstruktur in Produktdatensystemen und -archiven verwendbar. In der Praxis ist die Struktur von STEP allerdings von vorhandenen Produktdatensystemen inspiriert. Bei STEP wird die formale Spezifikationssprache EXPRESS verwendet, um Produktinformationen darzustellen. [UG96]
Es gibt andere Datenaustauschformate wie beispielsweise *Initial Graphics Exchange Specification* (IGES), die auf eine informelle Beschreibung zurückgreifen, um Inhalte mittels natürlichsprachlicher Texte oder Anwendungsbeispielen zu definieren. Das hat den Nachteil, dass eine Spezifikation mehrdeutig und unvollständig sein kann. Um diese Nachteile auszugleichen, wurde mit EXPRESS eine Spezifikationssprache für STEP verwendet, die sowohl zur Beschreibung von integrierten Ressourcen als auch zur Entwicklung von Anwendungsprotokollen eingesetzt wird. [ATJ+00]
In EXPRESS werden Klassen *Entities* genannt. Diese können neben Relationen zu anderen Entities auch Eigenschaften in Form von textuellen, numerischen oder logischen Variablen enthalten. Zusätzlich gibt es die Möglichkeit mit *Enumerations* eine Liste von festen Werten für einen Parameter vorzugeben. Neben der Darstellung der Beziehungen zwischen verschiedenen Entities und ihren Parametern in formeller Sprache, können diese auch mit der grafischen Notation *EXPRESS-G* gezeigt werden. In Abbildung 3.5 ist ein Beispiel eines einfachen Modells in EXPRESS und dessen grafische Darstellung mit EXPRESS-G dargestellt. Die beiden Entities *Bauteil* und *Geometrie* sind mit der Relation *Gestalt* miteinander verbunden. [ATJ+00]

EXPRESS

```
ENTITY Bauteil;
    Gestalt: Geometrie;
END_ENTITY;
ENTITY Geometrie;
END_ENTITY;
```

Abbildung 3.5: EXPRESS und EXPRESS-G Beispiel anhand zweier Entities *Bauteil* und *Geometrie*, die mit der Relation *Gestalt* verbunden sind nach [ATJ+00, Bild 4.3].

Zusätzlich existieren in EXPRESS lokale und globale Regeln, mit denen Zusammenhänge zwischen Entities klarer dargestellt werden können. Ebenso existiert die Möglichkeit Funktionen zu definieren, die beispielsweise die Masse einer kompletten Baugruppe berechnet, wobei eine Baugruppe alle unter einer bestimmten Entity befindlichen anderen Entities darstellt. [ATJ+00]
Neben der Implementierung mittels EXPRESS, die keinen logischen Formalismus hat, gibt es das Projekt *OntoSTEP*. Hierfür wurde vom EXPRESS Schema nach OWL-DL konvertiert und EXPRESS Instanzen als OWL Individuen klassifiziert. Durch diese Überführung kann auch formale Logik auf die Produktinformationen angewendet werden, wie beispielsweise [BKR+12]:

- **Konsistenzprüfung**

  Dieser Mechanismus stellt sicher, dass keine Widersprüche im Model vorhanden sind. Hierbei kann auf Schema- und auch Instanzebene nach Inkonsistenzen gesucht werden.

- **Schlussfolgerungen**

  Durch das Hinzufügen von logischen Zusammenhängen und den daraus resultierenden Schlüssen, kann neues Wissen gewonnen werden. Hier wurden *Reasoner* (Schlussfolgerer) implementiert, der funktionelle oder transitive Eigenschaften, Eigenschaftshierarchien und Eigenschaftsketten ausarbeitet.

- **Entscheidbarkeit**

  Das Ziehen von Schlussfolgerungen geschieht in endlicher Zeit. Durch die Verwendung der OWL-API als Schnittstelle zur Ontologie, über die Manipulationen und Anfragen getätigt werden können, können gezielt wichtige Informationen gefunden werden.

Das hier vorgestellte Format STEP (und auch OntoSTEP) stellt eine Zwischensprache in Form einer Ontologie dar, um Daten von einem System zu einem anderen zu bringen. Damit stellt STEP ein Beispiel für eine Migrationsontologie dar, die speziell im Automobil- und Luftfahrtbereich weite Verbreitung findet. Ähnlich der funktionalen Erweiterung OntoSTEP soll die hier entwickelte Ontologie Funktionen enthalten, die Anwendungen analysieren und darstellen. Damit stellt STEP und OntoSTEP einen guten Ausgangspunkt für die hier zu entwickelnde Ontologie dar, allerdings mit einem anderen Fokus.

3.4 Zusammenfassung

Es gibt eine Vielzahl von Anleitungen und Hilfsmitteln, um Migrationen zwischen verschiedenen Java EE Anwendungsservern zu erleichtern. Diese sind aber bis auf wenige Fälle auf je einen speziellen Ausgangs- und Zielserver festgelegt. Die wenigen Ausnahmen wie beispielsweise Windup (siehe Abschnitt 3.2.3) sind zwar nicht auf spezielle Ausgangsserver, dafür aber auf einen Zielserver beschränkt. Keines der angesprochenen Hilfsmittel ist derart erweiterbar, dass jeder mögliche Java EE Anwendungsserver unterstützt werden könnte.
Ontologien werden wie beispielsweise bei STEP als Hilfsmittel und Format für die Migration von Daten zwischen verschiedenen Systemen erfolgreich herangezogen. Dies ist allerdings noch nicht im Bereich der Java EE Anwendungen geschehen. Eine Repräsentation von Java EE Applikationen und deren Eigenschaften kann dabei helfen zu entscheiden, ob eine Anwendung auf einen anderen Server migriert werden kann oder nicht. Durch Bedingungen können direkt aus dieser Ontologie Schlussfolgerungen gezogen werden über die Migrierbarkeit einer Anwendung bzw. über den Aufwand für eine Migration.
In den nachfolgenden Kapiteln (4, 5, 6) wird eine Ontologie zur Repräsentation von Java EE Anwendungen entwickelt, mit deren Hilfe Migrationen zwischen verschiedenen Java EE Anwendungsservern erleichtert werden sollen.

4 Vorgehensweise und Prämissen der Ontologie

Dieses Kapitel beschäftigt sich mit Vorüberlegungen zur Entwicklung der Ontologie. Zuerst werden in Kapitel 4.1 unterschiedliche Methodiken zur Ontologieentwicklung vorgestellt. Kapitel 4.2 beschäftigt sich mit grundlegenden Anforderungen an die Ontologie selbst. Hier wird auch die hier verwendete Methodik zur Entwicklung der Ontologie festgelegt.

4.1 Methodik zur Entwicklung der Ontologie

Seit Beginn der 1990er Jahre steigt das Interesse der Wissenschaft und Wirtschaft an Ontologien deutlich an. Das Fehlen von gemeinsamen, strukturierten Richtlinien bremste die Entwicklung von Ontologien sehr stark ab. Deshalb begann man Prinzipien und praktische Vorschriften festzulegen, damit andere Ontologieentwickler davon profitieren konnten. Seitdem gibt es einige Methodiken, um den Ontologieentwicklungsvorgang zu vereinfachen und zu unterstützen. [GPCFL04]
Die Tätigkeit der Konstruktion oder Entwicklung einer Ontologie wird *Ontological Engineering* genannt. Nach [NM01] besteht das Entwickeln einer Ontologie darin, Klassen der Ontologie zu definieren, die Klassen in einer taxonomischen Hierarchie darzustellen, Attribute der Klassen zu definieren und anschließend die Ontologie mit Instanzen zu befüllen. Diese Vorgehensweise ist sehr stark vereinfacht dargestellt, gibt allerdings schon einen groben Überblick über die bei der Ontologieentwicklung zu erledigenden Schritte. [SFGPMG12]
Das Ziel dieses Kapitels ist die Vorstellung von einigen Methodiken zur Entwicklung einer Ontologie. In Abschnitt 4.2.4 wird die für diese Arbeit verwendete Methodik bestimmt.

4.1.1 Uschold und King

Uschold und King präsentierten die erste Methode zur Entwicklung einer Ontologie in [UK95], die dann von Uschold und Grüninger in [UG96] weiterentwickelt wurde. Hier werden einige Hinweise basierend auf ihrer Erfahrung beim Entwickeln der *Enterprise Ontology* gegeben. Die Hauptprozesse dieses Leitfadens sind nach [GPCFL04]:

1. **Identifizierung des Ziels und Rahmen der Ontologie**

   Das Ziel dieses Schritts ist es, herauszuarbeiten, warum die Ontologie entwickelt wird. Welchen Nutzen hat die Ontologie und was sind die relevanten Begriffe und Bestandteile des betrachteten Bereichs.

2. **Entwicklung der Ontologie**

   Dieser Bereich ist in die folgenden drei Teilaktivitäten aufgeteilt:

a) Aufnehmen der Ontologiebegriffe

Kernkonzepte und -relationen müssen identifiziert werden. Eine präzise und eindeutige textuelle Definition dieser Elemente muss erstellt werden. Es gibt hier drei Möglichkeiten zur Erlangung dieser Informationen:

- **bottom-up**

  Als erstes sollen die spezifischsten Bestandteile der Ontologie betrachtet werden. Anschließend werden dann die abstrakteren Teile definiert. Dieser Ansatz ist aufwändig, kann nur schwer Gemeinsamkeiten von Teilen erkennen, führt oft zu Inkonsistenzen und benötigt häufiges Überarbeiten.

- **top-down**

  Es wird mit den abstraktesten Bestandteilen begonnen und diese werden anschließend immer weiter spezifiziert. Dieser Ansatz kann zu abstrakten Konzepten führen, die willkürlich definiert und absolut unnötig sind.

- **middle-out**

  Bei diesem Ansatz werden zuerst die Kern- und Basisbegriffe definiert. Diese werden dann nach Belieben spezifiziert oder generalisiert. Nach [UG96] wird damit die richtige Balance zwischen Detaillierungsgrad und Abstraktion gefunden, was diesen Ansatz zur Methode der Wahl macht.

b) Implementieren der Ontologie

Diese Aufgabe beinhaltet neben der Implementierung auch das Festlegen von Basisbegriffen, die zur Spezifikation der Ontologie verwendet werden (also Klassen, Relationen, etc.).

c) Integration von existierenden Ontologien

In diesem Schritt werden schon vorhandene Ontologien in die neu entwickelte integriert, wo dies Sinn macht. Ontologien sind meist so gestaltet, dass sie wiederverwendbar sind.

3. Bewertung der Ontologie

Die Ontologie wird einer technischer Bewertung unterzogen, wobei auch grundsätzliche Anforderungen geprüft und mit der Ontologie verbundene Implementierungen unter die Lupe genommen werden.

4. Dokumentation der Ontologie

Die Ontologie wird dahingehend dokumentiert und veranschaulicht, dass allgemeine Kodierungsregeln gelten und andere Benutzer sie wiederverwenden können.

Selbst nach der eigenen Meinung der Autoren ist der hier kurz beschriebene Prozess keine vollständige Methodologie. Jede Methodologie muss Techniken, Methoden und Vorgehensbeschreibungen für die vier Schritte besitzen. Zusätzlich sollten Verbindungen wie Reihenfolge, Inputs oder Outputs beschrieben werden. [UG96]
Nach [GPCFL04] ist der Hauptkritikpunkt an dieser Methode der fehlende Konzeptualisierungsschritt vor der Implementierung. Das Ziel eines solchen Konzeptualisierungsschritts ist es ein weniger formelles Modell des betrachteten Bereichs zu erstellen. Durch das Fehlen

einer solchen für andere Menschen verständlichen Formulierung der Ontologie können Schwierigkeiten beim Verstehen der Ontologie entstehen. Nachträglich ein konzeptionelles Modell hinzuzufügen ist mit sehr viel Aufwand verbunden.

4.1.2 Grüninger und Fox

Gleichzeitig zu Uschold und King (siehe Abschnitt 4.1.1) entwickelten Grüninger und Fox [GF95] einen formalen Ansatz zur Entwicklung einer Ontologie und deren Evaluierung. Dieser Ansatz ist geprägt von der Entwicklung von wissensbasierten Systemen und verwendet Prädikatenlogik erster Stufe. In dieser Methode werden zuerst die Anwendungsfälle der Ontologie identifiziert und anschließend mit Kompetenzfragen der Umfang der Ontologie bestimmt. Diese Fragen und deren Antworten werden dazu verwendet, die Hauptkonzepte und deren Attribute, Relationen und formale Axiome der Ontologie zu ermitteln. Die folgenden Schritte werden in dieser Methodik unterschieden [GPCFL04]:

1. **Identifizierung der Anwendungsfälle**

   In diesem Schritt sollen alle Szenarien ermittelt werden, für die die Ontologie entwickelt wird, und alle Anwendungen, die die Ontologie verwenden werden.

2. **Ausarbeitung von informellen Kompetenzfragen**

   Mit den in Schritt 1 ermittelten informellen Szenarien sollen informelle Kompetenzfragen erarbeitet werden. Diese Fragen sollen von der Ontologie schließlich beantwortet werden können. Eine Ontologie ist nicht gut konstruiert, wenn alle Kompetenzfragen mit einfachen Anfragen an die Ontologie beantwortet werden können. Das bedeutet, dass die Fragen nicht zu spezifischeren oder allgemeineren Fragen zerfallen oder kombiniert werden können. Aus jeder Kompetenzfrage können beispielsweise Bedingungen, Annahmen oder notwendiger Input abgeleitet werden.

3. **Spezifizierung der Begrifflichkeiten mit Prädikatenlogik erster Stufe**

   Mit den Kompetenzfragen kann der Inhalt der Ontologie definiert werden. Dieser Inhalt wird in der Ontologie formell repräsentiert. Die Antworten der Kompetenzfragen dienen dazu, die formellen Definitionen von Konzepten, Relationen und Axiomen zu ermitteln.

4. **Umschreiben der Kompetenzfragen in formelle Terminologie**

   Die im zweiten Schritt definierten Kompetenzfragen werden in der Ontologiesprache übersetzt. Auf diese Art und Weise können sie leichter überprüft und von einem Computer ausgewertet werden. Kann eine Frage nicht in dieser Sprache verfasst werden, so ist die Ontologie nicht vollständig.

5. **Definition von Axiomen mit Prädikatenlogik erster Stufe**

   In diesem Schritt werden Konstanten und spezifische Definitionen der Ontologie festgelegt. Hier werden beispielsweise Bedingungen für Attribute definiert oder die formale Bedeutung von Relationen bestimmt. Wenn ein solches Axiom nicht in der Ontologiesprache dargestellt werden kann, muss die Ontologie erweitert werden.

6. **Bestimmung von Vollständigkeitssätzen**

   Für die mögliche Lösungen der Kompetenzfragen muss geprüft werden, unter welchen Bedingungen sie vollständig sind. Mit diesem Schritt wird die Ontologie evaluiert und deren Inhalt geprüft.

Diese Methodik führt die Schritte 1 bis 3 aus Abschnitt 4.1.1 aus und beschreibt damit die Identifizierung der Ziele der Ontologie, die Entwicklung der Ontologie und die Evaluierung sehr genau. Was auch hier fehlt sind Programme zur Unterstützung der Methodik. Daneben sind im Bereich der Management- und Unterstützungsprozesse keine Details erwähnt. [GPCFL04]

4.1.3 METHONTOLOGY

Die Methodik *METHONTOLOGY* wurde an der Universidad Politécnica de Madrid im Ontologie-Team von Asunción Gómez-Pérez und Mariano Fernández-López entwickelt. METHONTOLOGY basiert auf den Hauptaktivitäten des Softwareentwicklungsprozesses [Iee] und den Methodiken der Wissensingenieurwissenschaften [Wat85]. Inhalt dieser Methodologie sind die Identifikation von Ontologieentwicklungsprozessen, ein Lebenszyklus basierend auf sich weiterentwickelnden Prototypen und Techniken, um jede Aufgabe aus den Bereichen Management, Entwicklung und Unterstützung durchzuführen. Hierbei zählen zu den Managementaufgaben die Leitung der Entwicklung und die Qualitätssicherung. Zu den Entwicklungsprozessen zählen Spezifikation, Konzeptualisierung, Formalisierung, Implementierung und Wartung. Die Hilfsprozesse sind Informationsgewinnung, Integration, Evaluierung, Dokumentation und Konfigurationsmanagement. [FGJ97; Góm98; FGSS99]
METHONTOLOGY ist ein zyklischer Prozess, der als Ergebnis einen Prototypen besitzt, der mit jeder Iteration weiterentwickelt wird. In jedem Zyklus werden alle oben genannten Aktivitäten ausgeführt, wobei die Management- und Hilfsprozesse parallel zu den sequentiellen Entwicklungsprozessen ablaufen. Der in METHONTOLOGY wichtigste Prozess ist die Konzeptualisierung. Das liegt daran, dass dieser Schritt den Rest der Ontologieentwicklung festlegt. Das Ziel der Konzeptualisierung ist es das erhaltene Wissen so zu organisieren und strukturieren, damit es unabhängig von der verwendeten Ontologiesprache repräsentiert werden kann. Die Aktivität der Informationsgewinnung ist eng mit der Konzeptualisierung verknüpft. [GPCFL04]
METHONTOLOGY legt fest, dass die folgenden elf Aufgaben bei der Entwicklung einer Ontologie in der Konzeptualisierung durchgeführt werden müssen [GPCFL04]:

1. **Erstellen eines Glossars**

   Der Entwickler erstellt als erstes einen Glossar mit allen relevanten Begriffen des betrachteten Bereichs. Dazu gehören beispielsweise Konzepte/Klassen, Instanzen, Attribute und Relationen. Zusätzlich sollen auch Beschreibungen der Begriffe, Synonyme und Akronyme aufgenommen werden.

2. **Definieren einer Konzepttaxonomie**

   Sobald der Glossar eine gewisse Größe erreicht hat, sollte eine Konzepttaxonomie entwickelt werden. Hierfür können entweder bottom-up, top-down oder middle-out Strategien verwendet werden. Um eine solche Konzepttaxonomie zu erstellen, werden Begriffe aus dem Glossar als Konzepte definiert. In METHONTOLOGY wird vorgeschlagen die folgenden vier Taxonomierelationstypen zu verwenden:

a) Subclass-Of

Ein Konzept C_1 ist *Subclass-Of* eines anderen Konzepts C_2 genau dann, wenn jede Instanz von C_1 gleichzeitig eine Instanz von C_2 ist.

b) Disjoint-Decomposition

Eine *Disjoint-Decomposition* eines Konzepts C ist eine Menge von Subklassen von C, die keine gemeinsamen Instanzen besitzen und C nicht komplett abdecken.

c) Exhaustive-Decomposition

Eine *Exhaustive-Decomposition* eines Konzepts C ist eine Menge von Subklassen von C, die C komplett abdecken aber gemeinsame Instanzen besitzen können.

d) Partition

Eine *Partition* eines Konzepts C ist eine Menge von Subklassen von C, die C komplett abdecken und keine gemeinsamen Instanzen besitzen.

Wenn die Taxonomie erstellt ist, sollte sie auf Fehler wie Kreise in der Hierarchie oder gemeinsame Instanzen in einer Partition überprüft werden. Erst dann ist dieser Schritt beendet.

3. Konstruieren von binären Relationen

Nach der Taxonomie folgt in der Konzeptualisierungsphase das Erstellen von binären Relationsdiagrammen und damit das informelle Definieren dieser Relationen. Mit den Diagrammen werden Beziehungen zwischen einzelnen Konzepten dargestellt. Bevor mit dem nächsten Schritt weitergemacht werden kann, müssen die erstellten Diagramme auf ihre Richtigkeit überprüft werden. Dazu gehören vor allem die richtigen Konzepte an den Enden der Relationen und nicht deren Subklassen.

4. Erstellen des Konzeptwörterbuchs

In diesem Schritt werden die Attribute von Konzepten beschrieben. Optional können auch Instanzattribute definiert werden. Im hierfür geschriebenen Konzeptwörterbuch werden alle Konzepte, Relationen und Attribute aufgeführt und beschrieben.

Damit sind die Vorarbeiten der Ontologieentwicklung abgeschlossen und die Detaillierungsphase beginnt [GPCFL04]:

5. Detailliertes Definieren der binären Relationen

Ziel dieses Schritts ist es alle im Konzeptwörterbuch vorhandenen binären Relationen im Detail zu beschreiben. Das bedeutet, dass für jede Relation der Name, die Namen der Usprungs- und Zielkonzepte, die Kardinalität, die inverse Relation und die mathematischen Eigenschaften der Relation definiert werden müssen. Mathematische Eigenschaften sind hierbei beispielsweise Symmetrie oder Transitivität.

6. Detailliertes Definieren von Instanzattributen

Mit diesem Schritt sollen die Instanzattribute des Konzeptwörterbuchs im Detail definiert werden. Instanzattribute sind die Attribute, deren Werte sich bei verschiedenen Instanzen der selben Klasse unterscheiden können. Es müssen die folgenden Eigenschaften eines jeden Attributs definiert werden: sein Name; das Konzept zu dem es gehört;

der Typ des Werts; seine Maßeinheit, seine Genauigkeit und sein Wertebereich (im Fall von numerischen Werten); sein Standardwert; seine mögliche Liste von Werten (falls zutreffend); seine minimale und maximale Kardinalität; Formeln für die zugrundeliegenden Regeln hinter dem Attribut; Referenzen, mit denen das Attribut definiert wird. Am Ende dieses Schritts sollen alle Instanzattribute in der Instanzattributstabelle abgebildet sein.

7. **Detailliertes Definieren von Klassenattributen**

   Diese Phase hat das Ziel, dass die Klassenattribute im Detail definiert werden. Im Gegensatz zu Instanzattributen sind Klassenattribute Attribute, die auf allen Instanzen einer Klasse gleich sind. Für jedes dieser Attribute sind die folgenden Eigenschaften zu beschreiben: der Name des Attributs; der Name des zugehörigen Konzepts; der Typ des Attributswerts; der Attributswert; Maßeinheit und Genauigkeit (für numerische Werte); Kardinalität; die Instanzattribute, deren Werte von diesem Attribut abgeleitet sind. Das Ergebnis dieses Schritts ist die komplett befüllte Klassenattributstabelle.

8. **Detailliertes Definieren von Konstanten**

   In diesem Schritt werden die Konstanten aus dem Glossar bis ins Detail definiert. Jede Zeile in der Konstantentabelle beinhaltet eine detaillierte Beschreibung der Konstante. Folgende Eigenschaften müssen beschrieben werden: Name der Konstante; Typ des Konstantenwerts; Wert der Konstante; Maßeinheit (bei numerischen Konstanten); Attribute, die sich auf die Konstante beziehen.

Nach der Detaillierungsphase der Ontologieentwicklung werden anschließend Axiome und Regeln definiert. Diese sind wichtige Elemente bei der Ontologiemodellierung, da sie Wissen repräsentieren, das nicht mit normalen Komponenten dargestellt werden kann. Axiome sind logische Ausdrücke, die immer *true* zurückliefern. Sie werden dazu verwendet Bedingungen innerhalb der Ontologie zu definieren. Regeln können dazu benutzt werden, auf Informationen innerhalb einer Ontologie schlusszufolgern, wie beispielsweise Attributwerte oder Relationsinstanzen. In METHONTOLOGY ist es vorgesehen, Axiome und Regeln parallel nach den Schritten eins bis acht zu definieren. [GPCFL04]

9. **Definieren von Axiomen**

   In diesem Schritt müssen die in der Ontologie benötigten formalen Axiome identifiziert und genau beschrieben werden. Die folgenden Informationen müssen hierbei erbracht werden: Name des Axioms; informelle Beschreibung des Axioms; Logischer Ausdruck, der das Axiom in Prädikatenlogik erster Stufe definiert; Konzepte, Attribute und Relationen, auf die sich das Axiom bezieht; im Axiom verwendete Variablen. Für jedes solches Axiom wir in diesem Schritt in der Axiomtabelle ein Eintrag angelegt und befüllt.

10. **Definieren von Regeln**

    Ziel dieses Schritts ist es die in der Ontologie benötigten Regeln zu identifizieren und genau in der Regeltabelle zu beschreiben. Für jede einzelne Regel müssen dabei folgende Eigenschaften definiert werden: Name der Regel; informelle Beschreibung der Regel; Ausdruck, der die Regel formell beschreibt; Konzepte, Attribute und Relationen, auf die sich die Regel bezieht; in der Regel verwendete Variablen. METHONTOLOGY sieht vor den Regelausdruck als *if*-Statement zu beschreiben.

Optional können in einem elften Schritt Informationen über einzelne Instanzen in die Ontologie aufgenommen werden [GPCFL04]:

11. Definieren von Instanzen

Dieser optionale Schritt liefert Instanzen und definiert diese schon in der Ontologie, wenn sie wiederverwendet werden sollen.

Für die Methodik METHONTOLOGY können Programme wie *ODE*, *WebODE*, *Protégé-2000* oder *OntoEdit* verwendet werden. Diese Methodik wurde von der *Foundation for Intelligent Physical Agents (FIPA)* für die Entwicklung von Ontologien vorgeschlagen. FIPA fördert die gegenseitige Benutzbarkeit zwischen agentenbasierten Anwendungen. [GPCFL04]
Speziell den Schritt der Konzeptualisierung wird in METHONTOLOGY sehr detailliert beschrieben. Das liegt daran, dass die restlichen Aktivitäten sehr stark von den Ergebnissen dieses Schritts abhängen. Nach Beendigung der Konzeptualisierung wird das Konzeptmodell in ein formelles Modell transformiert, das im Implementierungsschritt in einem Programm umgesetzt wird. Hierbei wird im Laufe des Prozesses der Grad an Formalisierung des Modells in jedem Schritt erhöht. [GPCFL04; FGJ97; Góm98; FGSS99]
Zusammen mit den verwendbaren Programmen ist METHONTOLOGY eine sehr ausführlich beschriebene Methodik zum Erstellen einer Ontologie. Neben der Entwicklung der Ontologie geht METHONTOLOGY auch auf das Management und die Weiterentwicklung von Ontologien ein. Der Detailgrad der Phasen nach der Konzeptualisierung ist hierbei allerdings deutlich geringer als in der Konzeptionsphase.

4.2 Überlegungen zur Ontologie

Die hier entwickelte Ontologie soll eine Zwischensprache darstellen, mit der Eigenschaften von Applikationen repräsentiert werden können. Zusätzlich sollen unterschiedliche Java EE Anwendungsserver in der Ontologie als Instanzen von Backend-Systemen vorkommen, damit eine Prüfung auf Migrierbarkeit der Anwendung direkt in der Ontologie möglich ist. Diese Prüfung vergleicht die Eigenschaften eines Java EE Anwendungsservers mit denen der Applikation. Falls die Einstellungen der Anwendung mit dem Server kompatibel sind, ist eine Migration theoretisch möglich, andernfalls muss die Applikation für diesen Server angepasst werden.
Neben der Darstellung der Anwendung als Instanz in der Ontologie sollen auch die Analysefunktionen Bestandteil davon werden. Analysefunktionen sind in diesem Zusammenhang Methoden zur Analyse eines Anwendungspakets, um die Applikation mittels der Ontologie darzustellen. Diese Funktionen haben innerhalb der Ontologie Verknüpfungen zu den Klassen der Eigenschaften, die sie untersuchen sollen.
In diesem Kapitel soll dargelegt werden, warum eine Ontologie für diesen Anwendungsfall sinnvoll ist und welche Anforderungen an die zu entwickelnde Ontologie gestellt werden. Zusätzlich wird eine Bestandsaufnahme für die notwendigen Teile der Ontologie gegeben, was dann in Kapitel 5 detailliert wird.

4.2.1 Warum Ontologie?

Ontologien sind im Laufe der letzten 20 Jahre erfolgreich in sehr vielen Bereichen der Informationstechnologie eingesetzt worden. Speziell durch Zwischensprachen wie STEP oder

Hilfsmitteln bei Migrationen von Daten ist der Einsatzes von Ontologien mit Erfolg gezeigt worden. In diesem Bereichen werden Ontologien auch durch Weiterentwicklungen und Kombinationen mit anderen Technologien weiterhin ein große Rolle spielen. [UG96; ATJ+00; NM01; KR13a]
Eine einfache Taxonomie ist für die Migration von Java EE Anwendungen nicht ausreichend. Ontologien sind durch ihre Struktur mit Regeln und Bedingungen geradezu dafür prädestiniert anhand der enthalten Daten Rückschlüsse direkt ziehen zu können, ohne auf problematische Implementierungen zurückgreifen zu müssen. Diese Regeln können dabei einfacher angepasst und erweitert werden, als wenn starre Anwendungen überarbeitet werden müssten. In dieser Arbeit wurde sich also dafür entschieden eine Ontologie als Metasprache zur Java EE Applikationsmigration zu entwickeln. Es ist allerdings auch klar, dass die Entwicklung einer umfangreichen Ontologie sich als sehr komplex herausstellen kann. Aus diesem Grund wurde vor der Entwicklung der Ontologie eine sinnvolle Methodik dafür ausgewählt, um durch ein standardisiertes Verfahren Probleme und Mehraufwand zu vermieden.

4.2.2 Anforderungen an die Ontologie

In diesem Kapitel werden Anforderungen in Bezug auf Erweiterbarkeit, Standards, Programmiersprache, Struktur und Verständlichkeit an die Ontologie gestellt.

4.2.2.1 Erweiterbarkeit

Die wichtigste Eigenschaft der Ontologie muss die Erweiterbarkeit sein. Das liegt daran, dass in Zukunft immer neue Parameter, neue Server, neue Analysemethoden etc. existieren und gefunden werden, die auch integriert werden können und müssen, damit ein aussagekräftiges Ergebnis erhalten werden kann. Das Ziel dieser Arbeit ist es nicht eine für alle vorhandenen Server, in allen möglichen Versionen vollständige Ontologie zu schaffen, sondern die Ontologiestruktur so zu erstellen, dass neben ein paar Servern noch andere integriert werden können. Gleiches gilt für neue Konfigurationsparameter und analysierte Bereiche von Applikationen. Für diese Arbeit werden vorläufig nur die in Abschnitt 4.2.3 beschriebenen Elemente als Inhalt der Ontologie herangezogen. Mögliche zukünftige Erweiterungen werden in Kapitel 8.2 aufgezeigt.
Es kann in Ausnahmefällen auch möglich sein, manche Teilaspekte einer Anwendung nicht betrachten zu müssen, was dann dazu führt, dass diese Teile aus der Ontologie gestrichen werden können. Auch dieser Bereich fällt in den Bereich Erweiterbarkeit, wenn beispielsweise für manche Server veraltete Mechanismen nicht mehr betrachtet werden müssen.

4.2.2.2 Programmiersprache

Das Backend der Ontologie wird in Java implementiert. Durch die Betrachtung von Java EE Anwendungen und das notwendige genaue Betrachten der kompilierten Applikation muss Java auf dem Rechner vorhanden sein. Aus diesem Grund wird auch für das Backend Java herangezogen. Außerdem können auf relativ einfache Weise durch selbstgeschriebene Erweiterungen zur Analyse von Anwendungen Funktionen als Bibliotheken hinzugefügt werden, was die Erweiterbarkeit der Ontologie steigert.
Dies geschieht dadurch, dass Analysefunktionen in der Ontologie mit einer Java Klasse definiert werden, die in einem `jar`-Paket im Basisverzeichnis enthalten sind.

4.2.2.3 Ontologiesprache

Die Ontologie soll möglichst mit einer vorhandenen Ontologiesprache wie `rdf` oder `owl` modelliert werden. Gegebenenfalls wird die Repräsentation der Ontologie zur einfachen Erweiterung zu einem späteren Zeitpunkt nicht in dieser Sprache erfolgen, sondern in einem eigenen `xml` Schema, das die Begrifflichkeiten einfacher darstellen kann. Da das Hauptaugenmerk dieser Arbeit aber auf der Konstruktion der Ontologie beruht, wird hier nicht weiter auf diese vereinfachte Repräsentationssprache eingegangen.
Grundsätzlich sollen vorhandene Ontologiesprachen als Vorbild genommen werden. Das hat den Vorteil, dass für diese Sprachen Schnittstellen für Java vorhanden sind, mit denen bequem gearbeitet werden kann.

4.2.2.4 Verständlichkeit

Neben der Lesbarkeit des Ontologiemodells durch den Computer sollen auch Menschen in der Lage sein, enthaltene Inhalte der Ontologie zu verstehen. Da das mit zunehmender Komplexität nicht mehr erreichbar ist, wird in Kapitel 7 eine vereinfachte Darstellung der Ergebnisse mittels einer Auswertung ähnlich zu Abschnitt 3.2.3 implementiert.
Wichtig ist hierbei, dass Zusammenhänge leicht verständlich sind und durch manuelle Eingriffe erweitert werden können.

4.2.3 Bestandteile der Ontologie

Die in dieser Arbeit entwickelte Ontologie besteht aus den folgenden miteinander verknüpften Teilen:

- **Metasprache für Eigenschaften von Java EE Applikationen und Servern**

  Eine Java EE Anwendung soll in der Ontologie als Instanz mit Parametern repräsentiert werden, die die tatsächliche Anwendung widerspiegelt. Dazu gehören Einstellungen in Deployment Deskriptoren oder vorhandene und abhängige Klassen. Zusätzlich soll jeder betrachtete Java EE Applikationsserver ebenfalls als Instanz in der Ontologie vorhanden sein, mit jeweils eigenen möglichen Parametern und vorhandenen Klassen, die von Anwendungen verwendet werden können. Auf diese Art kann geprüft werden, ob die Anwendung prinzipiell auf dem Anwendungsserver lauffähig ist bzw. welche Einstellungen und Klassen nicht vorhanden sind.

- **Definition von Analysefunktionen zum Abbilden von Java EE Applikationen auf die Metasprache**

  Die Analyse der Java EE Anwendung spielt für die Migration eine sehr große Rolle. Hierfür müssen Funktionen definiert werden, die nach eigenen Vorgaben die Anwendung prüfen und deren Eigenschaften auf die Metasprache überführen. Ausschließlich die Informationen einer Anwendung sollen in der Metasprache abgebildet werden, die die Schnittstelle zum Anwendungsserver betreffen. Jede einzelne Funktion hat Relationen zu den analysierten Eigenschaften in der Ontologie und eine Implementierung, die die Funktionalität beinhaltet.

Bezüglich der Analysefunktionen ist noch anzumerken, dass es eventuell möglich ist, zusammen mit dem Anwendungscode und den Informationen der Metasprache eine Migration

durchzuführen. Dies ist aber nicht Teil dieser Arbeit. Hier steht die Analyse der Anwendungen und Repräsentation der gesammelten Informationen bezüglich der Migrierbarkeit im Vordergrund.

4.2.4 Festlegung der Methodik zur Entwicklung der Ontologie

In diesem Kapitel soll die Methodik für die Entwicklung der Ontologie festgelegt werden. Dazu werden die in Kapitel 4.1 vorgestellten Methodiken nochmals kurz zusammengefasst. Aus diesen Methodiken werden anschließend die für diese Arbeit sinnvollsten Elemente ausgewählt und in einer eigenen Methodik zusammengefasst.
Die Methodik von Uschold und King (vgl. Abschnitt 4.1.1) ist für die Gesamtentwicklung einer Ontologie geeignet, allerdings nicht sehr umfassend definiert. In den vier Hauptprozessen „Identifizierung des Ziels und Rahmen der Ontologie“, „Entwicklung der Ontologie“, „Bewertung der Ontologie“ und „Dokumentation der Ontologie“ wird nicht detailliert auf einzelne spezielle Aufgaben eingegangen. Es wird festgestellt, dass eine „middle-out“-Strategie oftmals besser als „bottom-up“ oder „top-down“ Methoden funktioniert und deshalb gewählt werden soll. Unabhängig davon ist die Struktur dieser Methodik sehr sinnvoll, muss aber noch mit präzisen Einzelaufgaben befüllt werden.
Die von Grüninger und Fox vorgestellte Methodik (vgl. Abschnitt 4.1.2) präzisiert die ersten drei Schritte der Uschold und King Methodik, lässt dafür aber die Dokumentation komplett wegfallen. In der Evaluationsphase („Bestimmung von Vollständigkeitssätzen“) wird mittels vorher definierten Kompetenzfragen der Funktionsumfang der entwickelten Ontologie überprüft und ausgewertet. Zusätzliche Unterstützungsprozesse sind nicht erwähnt und müssen noch genauer spezifiziert werden. Besonderer Wert in dieser Methodik wird auf die Kompetenzfragen gelegt, die neben der Evaluierung auch in allen anderen Phasen wichtig sind und die Gestalt der Ontologie wegen der gewünschten Ergebnisse stark beeinflussen.
Die mit METHONTOLOGY (vgl. Abschnitt 4.1.3) vorgestellte Methodik legt verstärkten Wert auf die Konzeptualisierung, also das Erstellen eines strukturierten Konzepts vor der eigentlichen Entwicklung der Ontologie. Dieser Prozess ist von hoher Bedeutung für die Gesamtentwicklung einer Ontologie, da hiermit das Grundgerüst festgelegt wird, aus dem später die formale Definition der Ontologie hervorgeht. Im Laufe dieses Prozesses wird der Grad der Formalisierung in jedem Schritt erhöht, was die Übersetzung in eine Ontologiesprache stark vereinfacht.
Die vorgestellten Methodiken unterscheiden sich nicht besonders in der wesentlichen Struktur („Konzept“, „Entwicklung“, „Evaluierung“ und „Dokumentation“). Vielmehr liegen die Unterschiede im Detaillierungsgrad und der unterschiedlichen Wichtigkeit der einzelnen Schritte.
Um die Schwachstellen der einzelnen Methodiken zu umgehen, wird hier folgende Kombination der vorhandenen Methodiken gewählt. Dabei werden die Phasen Entwicklung und Evaluation in Kapitel 6 zusammengefasst, da diese Teile sehr eng miteinander verbunden sind und die Implementierung der Ontologie direkt betreffen.

1. **Konzept**

   Die Ziele der Ontologie werden als Kompetenzfragen nach Grüninger und Fox definiert. Zusätzlich werden die elf Schritte von METHONTOLOGY angewendet und eine „middle-out“-Strategie eingesetzt. Diese Teile sind in Kapitel 5 beschrieben.

2. Entwicklung

Das Ergebnis der Konzeptphase wird anschließend mittels der Ontologiesprache OWL repräsentiert. Diese Implementierung wird in Kapitel 6.1 beschrieben.

3. Evaluierung

In Kapitel 6.2 werden die Kompentenzfragen anhand der implementierten Ontologie überprüft und gleichzeitig die Ontologie selbst verifiziert.

4. Dokumentation

Diese Arbeit stellt eine Grobdokumentation der Ontologie dar. Speziell Kapitel 5 beschreibt alle vorkommenden Elemente der Ontologie ausführlich.

5 Konzeptualisierung der Ontologie

In diesem Kapitel wird die Konzeptphase als erster der vier Schritte des Entwicklungsprozesses für eine Ontologie beschrieben. Zunächst werden Kompetenzfragen in Kapitel 5.1 nach Grüninger und Fox (vgl. Abschnitt 4.1.2) definiert, auf deren Grundlage die Ontologie beschrieben und anschließend auch evaluiert werden kann. Damit werden neben der Ausrichtung der Ontologie auch die generellen Ziele und Verwendungsmöglichkeiten abgesteckt.
Anschließend werden die Konzeptionalisierungsprozesse nach METHONTOLOGY (siehe Kapitel 5.2) durchgeführt, um die Begrifflichkeiten der Ontologie, Regeln und Relationen möglichst formal zu beschreiben.

5.1 Kompetenzfragen

Dieses Kapitel beschäftigt sich mit der Erstellung von sogenannten Kompetenzfragen (s. Abschnitt 4.1.2). Mit deren Hilfe können die Hauptkonzepte mit ihren Attributen, Relationen und Axiomen ermittelt werden. Hierbei ist zu beachten, dass nicht alle Kompetenzfragen nur einfache Anfragen an die Ontologie darstellen, sondern auch darüber hinausgehen. [GPCFL04]
Die Ontologie zur Unterstützung einer Migration von Java EE Applikationen hat einen speziellen Blickwinkel auf Java EE Anwendungen. Anwendungen werden aus der Sicht von Anwendungsserver und dem Java EE Standard betrachtet. In Kapitel 2 wurde auf den Java EE Standard und auf einige Anwendungsserver eingegangen. Aus diesen dort beschriebenen unterschiedlichen Anforderungen lassen sich grundlegende Bedingungen an eine Applikation ableiten, damit diese auf einem Server funktionsfähig ist:

Typ der Anwendung

Enterprise Anwendungen sind mit sehr hoher Wahrscheinlichkeit nur auf Anwendungsserver mit dem Java EE Full Profile lauffähig. Daher kann allein mit dem Typ einer Anwendung schon ein spezieller Servertyp ausgeschlossen werden. In der Ontologie wird dieser Aspekt mittels eines Parameters am Anwendungspaket realisiert.

Verwendete Java EE APIs

Jeder Server implementiert ein bestimmtes Profil des Java EE Standards, wobei nicht alle vorhandenen Java EE APIs von jedem Server unterstützt werden. Im Prinzip handelt es sich dabei um eine Erweiterung des vorhergehenden Punktes, wenn man von unterschiedlichen Anwendungstypen als verwendete Java EE APIs ausgeht. Die Java EE APIs werden über die verwendeten Klassen einer Anwendung in der Ontologie abgebildet und überprüft.

Java Klassen

Im Prinzip spiegeln die verwendeten Klassen einer Anwendung auch wieder, welche Java

EE APIs verwendet werden. Wenn einzelne Klassen nicht auf einem Server vorhanden sind, dann hat das natürlich auch zur Folge, dass die Anwendung dort nicht lauffähig ist. Jede Applikation verwendet in der Ontologie beliebig viele Java Klassen, deren Vorhandensein im Anwendungsserver überprüft werden muss.

Deployment Deskriptoren

In Kapitel 2.3 wird auf einige Servertypen eingegangen und deren spezifische, individuelle Deployment Deskriptoren. Andere Server unterstützen diese im Normalfall nicht beziehungsweise nur teilweise. Dieser Aspekt wird in der Ontologie über die Einstellungen und Parameter innerhalb eines Deployment Deskriptors überprüft. Da jede Eigenschaft eindeutig ist, ist das auch möglich.
Genauso wie einzelne Deployment Deskriptoren nicht von allen Servern unterstützt werden, werden auch dort vorgenommene Einstellungen nicht auf allen Servern unterstützt. Das kann beispielsweise an unterschiedlichen Java EE Versionen liegen. Jede Anwendung und jeder Anwendungsserver verwenden bzw. unterstützen nur eine begrenzte Auswahl von Einstellungen. Diese werden in der Ontologie als Eigenschaften an der Anwendung und dem Anwendungsserver repräsentiert und auf einander abgebildet, um die Kompatibilität zu prüfen.

Struktur

Aus der Dateistruktur eines Anwendungspakets lassen sich Schlüsse auf den Classpath und die wirklich erkannten Bibliotheken einer Anwendung ziehen. Der Classpath wird in unterschiedlichen Versionen des Java EE Standards teilweise anders ausgewertet, was auch zu Kompatibilitätsschwierigkeiten führt. Die Struktur von Anwendungspaketen wird in der Ontologie über die Verzeichnisse und Java Pakete und deren Inhalt abgebildet.

Auf weitere Informationen wird in dieser Arbeit nicht eingegangen, da das den Umfang deutlich überschreiten würde und mit den oberen Aspekten schon ein großer Teil der Probleme bei einer Migration abgedeckt werden. Alle hier beispielhaft aufgelisteten nicht betrachteten Aspekte einer Anwendung werden in Kapitel 8.2 nochmals kurz erwähnt:

jsp-Dateien

Bisher werden nur Java Klassen und deren verwendeten Java Klassen betrachtet. Das gleiche gilt natürlich auch für `jsp`-Dateien, also *Java Server Pages* Dateien, die zur Anzeige von Webseiten mit im Hintergrund liegender Java-Logik verwendet werden. Sie verwenden genauso Java Klassen, um zu funktionieren. Dieser Aspekt wird aufgrund von Zeitproblemen nicht betrachtet.

Java Klassen und Funktionen

Auf die von einzelnen Klassen verwendeten Funktionen und deren genaue Übergabeparameter wird in dieser Arbeit ebenfalls nicht eingegangen. Das vorrangige Ziel ist es so oder so zuerst auf das Vorhandensein der verwendeten Klasse zu prüfen.

Damit lassen sich die Anforderungen an die Ontologie in Form von Kompetenzfragen folgendermaßen formulieren. Anhand dieser Fragen wird der Umfang, die Funktionalität und die Struktur der Ontologie abgeleitet und inhaltlich zur Entwicklung der Ontologie beigetragen.

1. Welche Eigenschaften besitzt die analysierte Anwendung?
2. Wie sieht die Ordner- und Dateistruktur der analysierten Anwendung aus?
3. Welche Art von Anwendung ist die analysierte Applikation?
4. Aus welchen Arten von Anwendungen besteht die analysierte Applikation?
5. Welche Java EE APIs verwendet die Anwendung?
6. Welche Eigenschaften einer Anwendung sind mit einem speziellen Server kompatibel?
7. Welche Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?
8. Wie viele Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?
9. Kann die Anwendung auf einem speziellen Server funktionieren?
10. Auf welchen Anwendungsservern ist die analysierte Anwendung lauffähig?

Und die Kombination und vielleicht wichtigste Information aus den vorhergegangenen Fragen zusammen:

11. Welcher Server wäre der geeignetste für die analysierte Anwendung?

Nach der Konzeptualisierung der Ontologie im nächsten Kapitel und der schon sehr formalen Beschreibung der Aspekte, werden in Kapitel 5.3 die hier definierten Kompetenzfragen in der gleichen formalen Notation definiert. Anschließend werden mit diesen Kompetenzfragen das Modell der Ontologie anhand dieser Kompetenzfragen in Kapitel 6.2 evaluiert.

5.2 Prozesse nach METHONTOLOGY

In diesem Kapitel werden die Prozesse der Konzeptualisierung aus METHONTOLOGY für die hier entwickelte Ontologie durchgeführt. Diese elf Phasen werden in Abschnitt 4.1.3 kurz beschrieben und hier exakter für die Aspekte der Ontologie beispielhaft ausgeführt.

5.2.1 Erstellen eines Glossars

Nach [GPCFL04] wird zu Beginn der Konzeptualisierung ein Glossar erstellt, in dem alle relevanten Begriffe wie Klassen, Instanzen, Attribute und Relationen beschrieben werden. Zu jedem im Glossar beschriebenen Begriff wird eine kurze Beschreibung und der Typ angegeben. Ein Auszug aus der hier definierten Ontologie kann in Tabelle 5.1 betrachtet werden. In dieser Tabelle sind von jedem in der Ontologie verwendeten Typ genau zwei Beispiele exemplarisch aufgeführt. Die restlichen und präziseren Definitionen der einzelnen Konzepte erfolgt in den folgenden Kapiteln.

Tabelle 5.1: Auszug aus dem Glossar der entwickelten Ontologie.

Name	**Beschreibung**	**Typ**
Anwendungsserver	Ein Java EE Anwendungsserver, auf dem Java EE Anwendungen ausführbar sind	Klasse
Datei	Eine reale Datei im Dateisystem	Klasse
Class Path Element	Ein Element einer Java EE Anwendung, das auf dem jeweiligen Class Path liegt (*Applikation*, *Datei*)	Relation
unterstützt Eigenschaft	Welche Eigenschaft von einem Anwendungsserver unterstützt wird (*Anwendungsserver*, *Java EE Eigenschaft*)	Relation
Attributsname	Der Name eines Attributs	Instanzattribut
Package	Das Package einer *Java Klasse*	Instanzattribut
JAR_FILENAME	Die Dateiendung einer `jar`-Datei (`jar`)	Konstante
SERVER_STANDARD_REALISATION	Der Standardwert des Attributs *Ausführung* eines *Anwendungsservers*	Konstante
Oracle GlassFish v3.1.2.2 Full	Der Oracle GlassFish Server in Version 3.1.2.2 mit Full Profile	Instanz
Apache Tomcat v8.0.24	Der Apache Tomcat Server in Version 8.0.24	Instanz
Anwendungspaket gehört zu genau einer Applikation	Das Axiom, das besagt, dass jedem *Anwendungspaket* genau eine *Applikation* zugeordnet ist	Axiom
Java Paket Dateiname	Das Axiom, das besagt, dass der *Dateiname* eine *Java Pakets* mit `war`, `ear`, `rar` oder `jar` enden muss	Axiom
Typ eines Java Pakets einer Webanwendung	Die Regel, dass das *Anwendungspaket* einer *Webanwendung* den *Typ* `war` besitzen muss	Regel
Webanwendung wegen Typ war	Die Regel, die besagt, dass eine *Applikation* mit dem *Typ* `war` des zugehörigen *Anwendungspakets* eine *Webanwendung* sein muss	Regel

5.2.2 Definieren einer Klassentaxonomie

Nachdem das Glossar erstellt wurde, muss eine Klassentaxonomie erzeugt werden. Hierbei ist zwischen den unterschiedlichen Vererbungsarten *Subclass-Of*, *Disjoint-Decomposition*, *Exhaustive-Decomposition* und *Partition* zu unterscheiden (vgl. Abschnitt 4.1.3). Die einfachste Repräsentation dieser Taxonomie erfolgt über Vererbungsdiagramme. Es ist besonders darauf zu achten, dass keine Fehler wie Schleifen in der Taxonomie vorkommen. [GPCFL04] In diesem Kapitel werden alle Vererbungen aus der entwickelten Ontologie vorgestellt (vgl. Abbildungen 5.1, 5.2, 5.3 und 5.4).
Abbildung 5.1 zeigt den Auszug aus der Klassentaxonomie der Ontologie für die Superklasse *Applikation*. Die Klasse *Applikation* besitzt die *Partition* bestehend aus *Webanwendung*, *Res-*

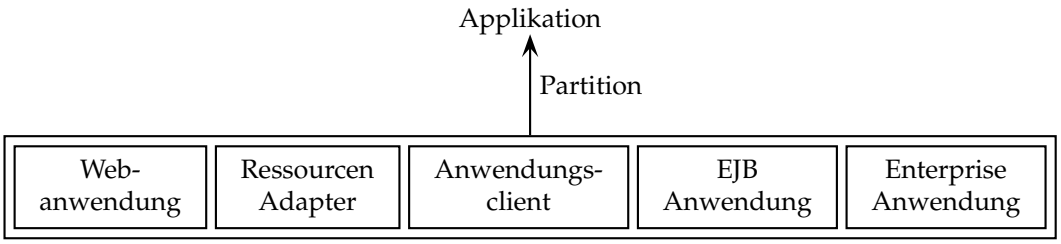


Abbildung 5.1: Der Auszug aus der Klassentaxonomie für die Superklasse *Applikation* und ihre Subklassen.

sourcenadapter, *Anwendungsclient*, *EJB Anwendung* und *Enterprise Anwendung*. Das bedeutet, dass jede Instanz von *Applikation* entweder eine *Webanwendung*, ein *Ressourcenadapter*, ein *Anwendungsclient*, eine *EJB Anwendung* oder eine *Enterprise Anwendung* ist.

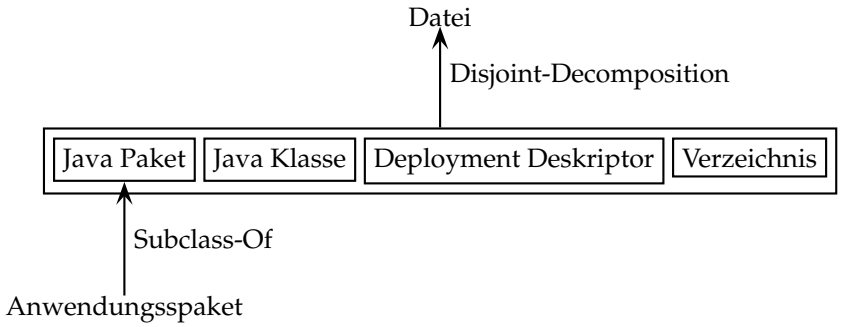


Abbildung 5.2: Der Auszug aus der Klassentaxonomie für die Superklasse *Datei* und ihre Subklassen.

In Abbildung 5.2 ist der Auszug aus der Klassentaxonomie der Ontologie für die Superklasse *Datei* abgebildet. Die Klasse *Datei* besitzt eine *Disjoint-Decomposition* bestehend aus *Java Paket*, *Java Klasse*, *Deployment Deskriptor* und *Verzeichnis*. Das bedeutet, dass *Java Paket*, *Java Klasse*, *Deployment Deskriptor* und *Verzeichnis* Subklassen von *Datei* sind, die keine gemeinsamen Instanzen besitzen können, aber *Datei* nicht vollständig beschreiben. Ein *Anwendungspaket* ist ein spezielles *Java Paket*, das auf einem *Anwendungsserver* ausgeführt wird.

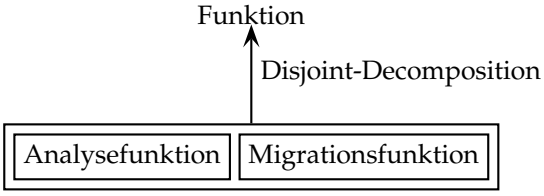


Abbildung 5.3: Der Auszug aus der Klassentaxonomie für die Superklasse *Funktion* und ihre Subklassen.

Abbildung 5.3 zeigt die Klassentaxonomie für die Klasse *Funktion*. Diese Klasse hat die beiden Subklassen *Analysefunktion* und *Migrationsfunktion*, die eine *Disjoint-Decomposition* darstellen. Eine *Analysefunktion* soll in der abschließenden Implementierung eine Anwendung

analysieren und deren Eigenschaften auf die Ontologie abbilden. Vorläufig sind aufgrund der Komplexität einer Migration keine Instanzen der Klasse *Migrationsfunktion* vorgesehen, sie sind aber dennoch schon als separate Klassen in der Ontologie aufgeführt.

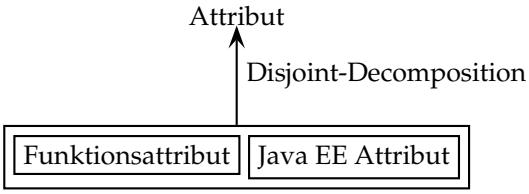


Abbildung 5.4: Der Auszug aus der Klassentaxonomie für die Superklasse *Attribut* und ihre Subklassen.

In Abbildung 5.4 ist die Klassentaxonomie für die Klasse *Attribut* abgebildet. Die beiden Klassen *Funktionsattribut* und *Java EE Attribut* bilden eine *Disjoint-Decomposition* der Klasse *Attribut*. Es wird zwischen diesen beiden Klassen unterschieden, damit klar ist zu welcher Art von Objekt das jeweilige Attribut gehört. *Funktionsattribute* gehören zu *Funktionen*, wohingegen *Java EE Attribute* zu *Java EE Eigenschaften* gehören.

5.2.3 Konstruieren von binären Relationen

Nach der Erstellung und Evaluierung der Klassentaxonomie müssen binäre Relationen definiert werden. Das Ziel der hier erstellten Diagramme ist es, die Verbindung zwischen unterschiedlichen Klassen der Ontologie darzulegen. In diesem Schritt können Fehler dann auftreten, wenn die Domäne oder der Umfang von Klassenzusammengehörigkeiten ungenau oder übergenau spezifiziert werden. [GPCFL04]
In diesem Kapitel werden beispielhaft einige Relationen der Ontologie in zwei Diagrammen (vgl. Abbildung 5.5 und Abbildung 5.6) gezeigt. Eine genauere Definition aller Relationen der Ontologie is in Abschnitt 5.2.5 ausgearbeitet.

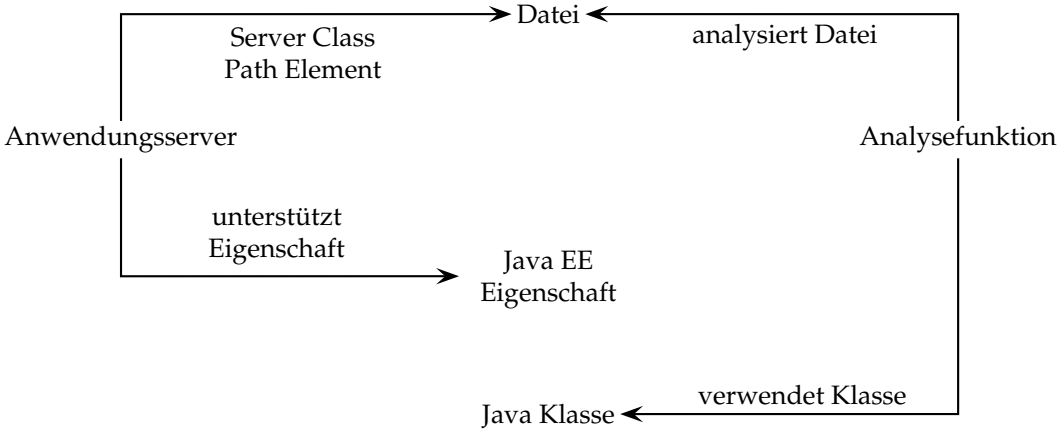


Abbildung 5.5: Ein Auszug aus dem Diagramm der binären Relationen für die Klassen *Datei*, *Anwendungsserver*, *Analysefunktion* und *Java EE Eigenschaft*.

In Abbildung 5.5 sind einige Relationen zwischen verschiedenen Klassen aufgezeigt, die teilweise auch schon in Tabelle 5.1 erwähnt wurden. Die Relation *Server Class Path Element*

ausgehend von *Anwendungsserver* zu *Datei* beschreibt welche *Datei* bei einem *Anwendungsserver* auf dem Class Path liegt und von einer Anwendung verwendbar ist, dazu kann natürlich auch ein *Java Paket* oder eine *Java Klasse* gehören. Hierbei ist zu beachten, dass ein *Anwendungsserver* beliebig viele Relationen vom Typ *Server Class Path Element* zu unterschiedlichen Instanzen vom Typ *Datei* besitzen kann.
Ein *Anwendungsserver* besitzt auch beliebig viele Relationen vom Typ *unterstützt Eigenschaft* zu je einer Instanz der Klasse *Java EE Eigenschaft*. Die Klasse *Analysefunktion* besitzt die Relation *analysiert Datei* zu Instanzen der Klasse *Datei*. Die Relation *verwendet Klasse* zu Instanzen der Klasse *Java Klasse* ist eigentlich auf die Superklasse *Funktion* von *Analysefunktion* definiert, wird aber auf *Analysefunktion* weiter vererbt.

Abbildung 5.6: Ein Auszug aus dem Diagramm der binären Relationen für die Klassen *Datei*, *Java Paket* und *Verzeichnis*.

Abbildung 5.6 zeigt die Relationen *ist Paketbestandteil von* und *ist Verzeichnisbestandteil von* der Klasse *Datei*, die auf die Klassen *Java Paket* respektive *Verzeichnis* verweisen. Jede *Datei* kann maximal eine dieser beiden Relationen besitzen, da jede reale Datei auch nur entweder in einem Verzeichnis oder einem Paket auf unterster Ebene enthalten sein kann. Instanzen der Klassen *Java Paket* und *Verzeichnis* können prinzipiell beliebig viele Bestandteile besitzen. Da *Java Paket* und *Verzeichnis* gleichzeitig zur Klasse *Datei* gehören, können sie auch Ursprung der Relationen *ist Paketbestandteil von* oder *ist Verzeichnisbestandteil von* sein.

5.2.4 Erstellen des Klassenwörterbuchs

Nachdem die Klassentaxonomie und das Diagramm der binären Relationen erstellt wurden, folgt die Definition von Parametern und Relationen der Klassen und deren Instanzen in einem Klassenwörterbuch. Dieses beinhaltet alle Klassen, ihre Relationen, ihre Instanzen und deren Klassen- und Instanzattribute. Relationen werden der Klasse zugeordnet, von der sie ausgehen. [GPCFL04]
Hierbei ist zu beachten, dass Relationen, Instanz- und Klassenattribute lokal nur einer einzelnen Klasse gehören, also bei anderen Klassen wiederverwendet werden dürfen. Aus Gründen der Übersichtlichkeit wird hier jedoch davon Abstand genommen. Ein Beispiel hierfür sind die Relationen *ist Verzeichnisbestandteil von* und *ist Paketbestandteil von*, die von der Klasse *Datei* entweder zu *Verzeichnis* oder *Java Paket* reichen. Die Relation *Datei direkt enthalten in* ist eine Relation von *Datei* zu *Datei* und ist nur dann vorhanden, wenn entweder *ist Paketbestandteil von* oder *ist Verzeichnisbestandteil von* vorhanden ist. [GPCFL04]

Tabelle 5.2: Klassenwörterbuch der entwickelten Ontologie.

Klassenname	**Instanzattribute**	**Relationen**
Applikation	Applikationsname	ist Teilanwendung von, Teilanwendung, besitzt Anwendungspaket, verwendet Eigenschaft, Class Path Element
Webanwendung	–	–
Ressourcen Adapter	–	–
Anwendungsclient	–	–
EJB Anwendung	–	–
Enterprise Anwendung	–	–
Datei	Dateiname	ist Paketbestandteil von, ist Verzeichnisbestandteil von, ist Server Class Path Element von, ist analysierte Datei von, ist Class Path Element von, Datei direkt enthalten in, Datei enthalten in, enthält Datei direkt, enthält Datei
Java Paket	Typ	Paketbestandteil
Anwendungspaket	–	ist Anwendungspaket von
Java Klasse	Name, Package	benötigt Klasse, ist benötigte Klasse von, wird verwendet von
Deployment Deskriptor	–	–
Verzeichnis	–	Verzeichnisbestandteil
Funktion	Funktionsname	besitzt Funktionsattribut, verwendet Klasse
Analysefunktion	–	analysiert Datei
Migrationsfunktion	–	–
Attribut	Attributsname, Attributswert	–
Funktionsattribut	–	ist Funktionsattribut von
Java EE Attribut	–	ist Java EE Attribut von, Java EE Attribut kompatibel mit
Anwendungsserver	Servername Version, Ausführung	Server Class Path Element, unterstützt Eigenschaft, unterstützt Java EE Attribut

Tabelle 5.2: Klassenwörterbuch der entwickelten Ontologie. – Fortsetzung

Klassenname	**Instanzattribute**	**Relationen**
Java EE Eigenschaft	Eigenschaftsname	ist Teileigenschaft von, Teileigenschaft, ist unterstützte Eigenschaft von, besitzt Java EE Attribut, ist verwendete Eigenschaft von, Applikation Eigenschaft entspricht, Anwendungsserver Eigenschaft entspricht

Tabelle 5.2 zeigt das Klassenwörterbuch der entwickelten Ontologie. Hierbei wurden Klassenattribute weggelassen, weil es in unserem Fall keine Klassen- sondern ausschließlich Instanzattribute gibt.
Nach dem Erstellen des Klassenwörterbuchs müssen die binären Relationen, Klassen- und Instanzattribute im Detail definiert werden. Zusätzlich sollen alle Konstanten exakt beschrieben werden. Hierbei ist die Reihenfolge nicht essenziell, in dieser Arbeit wird jedoch nach der in [GPCFL04] vorgestellten Reihenfolge vorgegangen.

5.2.5 Detailliertes Definieren der binären Relationen

In diesem Schritt werden alle binären Relationen, die im Klassenwörterbuch auftreten in der Tabelle der binären Relationen definiert. Für jede solche Relation müssen der Name, die Ursprungs- und Zielklasse, die jeweilige Kardinalität, die Gegenrelation und die mathematischen Eigenschaften angegeben werden. In dieser Arbeit werden die mathematischen Eigenschaften weggelassen, da keine der verwendeten Relationen symmetrisch, transitiv oder reflexiv ist. Zusätzlich wird nicht auf die Zielkardinalität eingegangen, da jede Relation immer auf genau eine Instanz einer Klasse zeigt. [GPCFL04]

Tabelle 5.3: Tabelle der binären Relationen der entwickelten Ontologie.

Relationsname	**Ursprungsklasse**	**Ursprungskardinalität**	**Zielklasse**	**Gegenrelation**
ist Teilanwendung von	Applikation	0–1	Applikation	Teilanwendung
Teilanwendung	Applikation	0–*	Applikation	ist Teilanwendung von
besitzt Anwendungspaket	Applikation	1	Anwendungspaket	ist Anwendungspaket von
verwendet Eigenschaft	Applikation	0–*	Java EE Eigenschaft	ist verwendete Eigenschaft von
Class Path Element	Applikation	0–*	Datei	ist Class Path Element von

Tabelle 5.3: Tabelle der binären Relationen der entwickelten Ontologie. – Fortsetzung

Relationsname	**Ursprungsklasse**	**Ursprungskardinalität**	**Zielklasse**	**Gegenrelation**
ist Paketbestandteil von	Datei	0–1	Java Paket	Paketbestandteil
ist Verzeichnisbestandteil von	Datei	0–1	Verzeichnis	Verzeichnisbestandteil
ist Server Class Path Element von	Datei	0–*	Anwendungsserver	Server Class Path Element
ist analysierte Datei von	Datei	0–*	Analysefunktion	analysiert Datei
ist Class Path Element von	Datei	0–*	Applikation	Class Path Element
Datei direkt enthalten in	Datei	0–1	Datei	enthält Datei direkt
Datei enthalten in	Datei	0–*	Datei	enthält Datei
enthält Datei direkt	Datei	0–*	Datei	Datei direkt enthalten in
enthält Datei	Datei	0–*	Datei	Datei enthalten in
Paketbestandteil	Java Paket	0–*	Datei	ist Paketbestandteil von
ist Anwendungspaket von	Anwendungspaket	1	Applikation	besitzt Anwendungspaket
benötigt Klasse	Java Klasse	0–*	Java Klasse	ist benötigte Klasse von
ist benötigte Klasse von	Java Klasse	0–*	Java Klasse	benötigt Klasse
wird verwendet von	Java Klasse	0–*	Analysefunktion	verwendet Klasse
Verzeichnisbestandteil	Verzeichnis	0–*	Datei	ist Verzeichnisbestandteil von
besitzt Funktionsattribut	Funktion	0–*	Funktionsattribut	ist Funktionsattribut von
verwendet Klasse	Funktion	1	Java Klasse	wird verwendet von
analysiert Datei	Analysefunktion	0–*	Datei	ist analysierte Datei
ist Funktionsattribut von	Funktionsattribut	0–1	Funktion	besitzt Funktionsattribut
ist Java EE Attribut von	Java EE Attribut	0–1	Java EE Eigenschaft	besitzt Java EE Attribut
Java EE Attribut kompatibel mit	Jave EE Attribut	0–*	Anwendungsserver	unterstützt Java EE Attribut

Tabelle 5.3: Tabelle der binären Relationen der entwickelten Ontologie. – Fortsetzung

Relationsname	**Ursprungsklasse**	**Ursprungskardinalität**	**Zielklasse**	**Gegenrelation**
Server Class Path Element	Anwendungsserver	0–*	Datei	ist Server Class Path Element von
unterstützt Eigenschaft	Anwendungsserver	0–*	Java EE Eigenschaft	ist unterstützte Eigenschaft von
unterstützt Java EE Attribut	Anwendungsserver	0–*	Java EE Attribut	Java EE Attribut kompatibel mit
ist Teileigenschaft von	Java EE Eigenschaft	0–1	Java EE Eigenschaft	Teileigenschaft
Teileigenschaft	Java EE Eigenschaft	0–*	Java EE Eigenschaft	ist Teileigenschaft von
ist unterstützte Eigenschaft von	Java EE Eigenschaft	0–*	Anwendungsserver	unterstützt Eigenschaft
besitzt Java EE Attribut	Java EE Eigenschaft	0–*	Java EE Attribut	ist Java EE Attribut von
ist verwendete Eigenschaft von	Java EE Eigenschaft	0–*	Applikation	verwendet Eigenschaft
Applikation Eigenschaft entspricht	Java EE Eigenschaft	0–*	Java EE Eigenschaft	Anwendungsserver Eigenschaft entspricht
Anwendungsserver Eigenschaft entspricht	Java EE Eigenschaft	0–*	Java EE Eigenschaft	Applikation Eigenschaft entspricht

In Tabelle 5.3 sind alle Relationen der entwickelten Ontologie aufgeführt. Die Werte der Ursprungskardinalität sind in folgender Aufzählung erklärt:

- **0–1**

  Jede Ursprungsklasse kann entweder keine oder genau eine dieser Relationen besitzen.

- **0–***

  Jeder Ursprungsklasse kann beliebig viele Relationen diesen Typs besitzen, auch keine.

- **1**

  Jeder Ursprungsklasse kann nur genau eine dieser Relationen besitzen.

Anzumerken hierbei ist noch, dass eine *Datei* entweder die Relation *ist Paketbestandteil von* oder *ist Verzeichnisbestandteil von* besitzen kann. Beides gleichzeitig ist nicht möglich.
Für *Anwendungsserver* sind mehrere *Java EE Attribute* mit gleichem Namen für eine *Java EE Eigenschaft* zulässig, um mögliche unterschiedliche Werte abzubilden. Bei einer *Applikation* ist das nicht der Fall, hier darf jede *Java EE Eigenschaft* auch jeweils nur ein *Java EE Attribut*

mit dem gleichen Namen besitzen. In der Ontologie werden gleiche *Java EE Eigenschaften*, die zu unterschiedlichen *Anwendungsservern* oder *Applikationen* gehören immer als eigene Instanzen aufgeführt. Mit der Relation *Applikation Eigenschaft entspricht* kann dann geprüft werden, ob diese *Java EE Eigenschaft* einer *Applikation* auch für einen *Anwendungsserver* vorhanden ist. Die Relation *Java EE Attribut kompatibel mit* beschreibt, welches *Java EE Attribut* auf einem *Anwendungsserver* unterstützt wird.

5.2.6 Detailliertes Definieren von Instanzattributen

In diesem Schritt werden alle im Klassenwörterbuch enthaltenen Instanzattribute mittels der Tabelle der Instanzattribute im Detail definiert. Jede Zeile dieser Tabelle liefert eine detaillierte Beschreibung eines Instanzattributs. Instanzattribute sind Parameter, deren Werte bei jeder Instanz unterschiedlich sein können. Für jedes solche Attribut müssen die folgenden Informationen gegeben werden [GPCFL04]:

- Name
- zugehörige Klasse
- Wertetyp
- Maßeinheit des Werts
- Genauigkeit des Werts
- Wertebereich
- Standardwert
- Kardinalität (minimal und maximal)
- Instanzattribute, Klassenattribute und Konstanten, aus denen der Wert abgeleitet wird
- Attribute, die durch dieses Instanzattribut abgeleitet werden können
- Formeln oder Regeln, die den Attributswert festlegen
- Referenzen, die das Attribut festlegen

Die mathematischen Eigenschaften wie Maßeinheit oder Genauigkeit des Wertes spielen in der hier entwickelten Ontologie keine Rolle, da es keine numerischen Attributswerte gibt. Ebenso ist der Wertetyp immer ein String. Die Kardinalität jedes hier beschriebenen Attributs ist immer genau eins.

Tabelle 5.4: Tabelle der Instanzattribute der entwickelten Ontologie.

Name	**Klasse**	**Standard-wert (Konstante)**	**Wertebereich (Konstante)**	**Regeln**
Applikations-name	Applikati-on			

Tabelle 5.4: Tabelle der Instanzattribute der entwickelten Ontologie. – Fortsetzung

Name	**Klasse**	**Standard-wert (Konstante)**	**Wertebereich (Konstante)**	**Regeln**
Dateiname	Datei			Java Paket Dateiname, Dateiendung war bedingt Typ des Java Pakets
Typ	Java Paket	jar (JAR_ FILENAME)	jar (JAR_ FILENAME), war (WAR_ FILENAME), ear (EAR_ FILENAME), rar (RAR_ FILENAME)	Typ eines Java Pakets einer Webanwendung, Webanwendung wegen Typ war, Dateiendung war bedingt Typ des Java Pakets, Typ war eines Java Pakets erzwingt Anwendungspaket
Name	Java Klasse			
Package	Java Klasse			
Funktionsname	Funktion			
Attributsname	Attribut			Kompatibilität eines Java EE Attributs
Attributswert	Attribut			Kompatibilität eines Java EE Attributs
Servername	Anwen-dungsser-ver			
Version	Anwen-dungsser-ver			
Ausführung	Anwen-dungsser-ver	Standard (SERVER_ STANDARD_ REALISA TION)		

Tabelle 5.4: Tabelle der Instanzattribute der entwickelten Ontologie. – Fortsetzung

Name	**Klasse**	**Standard-wert (Konstante)**	**Wertebereich (Konstante)**	**Regeln**
Eigenschaftsname	Java EE Eigenschaft			Applikation Java EE Eigenschaft entspricht, Applikation Java EE Eigenschaft auf oberster Ebene entspricht

In Tabelle 5.4 Tabelle der Instanzattribute der entwickelten Ontologie dargestellt. Die hier aufgeführten Regeln und Axiome sind in den Kapiteln 5.2.9 und 5.2.10 genau beschrieben. Für das Instanzattribut *Typ* sind aus Platzgründen nur die Regeln und Axiome des Werts `war` erwähnt. Auf die fehlenden Regeln und Axiome wird in den entsprechenden Kapiteln (5.2.9 und 5.2.10) genauer eingegangen.

5.2.7 Detailliertes Definieren von Klassenattributen

Klassenattribute sind Attribute, die auf allen Instanzen einer Klasse gleich sind. In der hier entwickelten Ontologie existieren keine separaten Klassenattribute, sondern ausschließlich Instanzattribute. Aus diesem Grund wird in diesem Kapitel auch keine Tabelle mit entsprechenden Attributen gezeigt.

5.2.8 Detailliertes Definieren von Konstanten

In diesem Schritt sollen die im Glossar der Ontologie enthaltenen Konstanten genau definiert werden. Dafür ist in der Konstantentabelle in jeder Zeile eine Konstante mit den folgenden Informationen zu spezifizieren:

- Name der Konstante
- Typ des Konstantenwerts
- Wert der Konstante
- Maßeinheit (bei nummerischen Konstanten)
- Attribute, die auf den Wert der Konstante zurückgreifen

Tabelle 5.5 zeigt die Konstantentabelle für die Konstanten *JAR_FILENAME*, *WAR_FILE NAME*, *EAR_FILENAME*, *RAR_FILENAME* und *SERVER_STANDARD_REALISATION*. Diese sind alle vom Typ String. Die ersten vier werden bei den Attributen *Dateiname* und *Typ* der Klasse *Java Paket* verwendet. *SERVER_STANDARD_REALISATION* bezieht sich auf das Attribut *Ausführung* der Klasse *Anwendungsserver*.

Tabelle 5.5: Konstantentabelle der entwickelten Ontologie.

Name	**Werte Typ**	**Wert**	**verwendete Attribute**
JAR_FILENAME	String	jar	Dateiname, Typ
WAR_FILENAME	String	war	Dateiname, Typ
EAR_FILENAME	String	ear	Dateiname, Typ
RAR_FILENAME	String	rar	Dateiname, Typ
SERVER_STANDARD_REALISATION	String	Standard	Ausführung

5.2.9 Definieren von Axiomen

In diesem Schritt, müssen die formalen Axiome identifiziert und detailliert beschrieben werden. Für jedes formale Axiom sollten nach METHONTOLOGY die folgenden Informationen gegeben werden [GPCFL04]:

- Name des Axioms
- Beschreibung des Axioms in normaler Sprache
- logischer Ausdruck, der das Axiom formal in Prädikatenlogik erster Stufe beschreibt
- die Klassen, Attribute und Relationen, auf die sich das Axiom bezieht
- die verwendeten Variablen

In [GPCFL04] wird zur Beschreibung des logischen Ausdrucks eines Axioms oder einer Regel eine Notation der Prädikatenlogik erster Stufe verwendet, die auch in dieser Arbeit verwendet wird. In Tabelle 5.6 sind die Bestandteile und Bedeutungen der verwendeten Notation kurz aufgeführt.

Tabelle 5.6: Erläuterungen zur Notation der hier verwendeten logischen Ausdrücke in Prädikatenlogik erster Stufe.

Notation	**Erläuterung**
`[Klasse](?X)`	Die Variable `?X` ist eine Instanz der Klasse *Klasse*
`[Relation](?X, ?Y)`	Die Variablen `?X` und `?Y` sind über die Relation *Relation* verbunden
`(?X)[Attribut]`	Die Variable `?X` besitzt das Attribut *Attribut*

Jedes Axiom wird in der Tabelle der formalen Axiome beschrieben. Das Axiom *Applikation besitzt genau ein Anwendungspaket* (Tabelle 5.7) beschreibt, dass jeder *Applikation* genau ein *Anwendungspaket* über die Relation *besitzt Anwendungspaket* zugeordnet werden muss.

Tabelle 5.7: Auszug aus der Tabelle der formalen Axiome der entwickelten Ontologie für das Axiom *Applikation besitzt genau ein Anwendungspaket.*

Name des Axioms	Applikation besitzt genau ein Anwendungspaket
Beschreibung	Jeder Applikation muss genau ein Anwendungspaket zugeordnet sein

Ausdruck	`forall(?X)`
	`([Applikation](?X) and`
	`[besitzt Anwendungspaket](?X, ?Y) and`
	`[Anwendungspaket](?Y))`

Klassen	Applikation, Anwendungspaket
Attribute	–
Konstanten	–
Relationen	besitzt Anwendungspaket
Variablen	?X, ?Y

Tabelle 5.8: Auszug aus der Tabelle der formalen Axiome der entwickelten Ontologie für das Axiom *Anwendungspaket gehört zu genau einer Applikation.*

Name des Axioms	Anwendungspaket gehört zu genau einer Applikation
Beschreibung	Jedem Anwendungspaket ist genau eine Applikation zugeordnet

Ausdruck	`forall(?X)`
	`([Anwendungspaket](?X) and`
	`[Applikation](?Y) and`
	`[ist Anwendungspaket von](?X, ?Y))`

Klassen	Anwendungspaket, Applikation
Attribute	–
Konstanten	–
Relationen	ist Anwendungspaket von
Variablen	?X, ?Y

Das Axiom *Anwendungspaket gehört zu genau einer Applikation* aus Tabelle 5.8 beschreibt, dass zu jeder Instanz der Klasse *Anwendungspaket* immer eine Instanz der Klasse *Applikation* über die Relation *ist Anwendungspaket von* zugeordnet werden muss. Das ist die Umkehrung des Axioms *Applikation besitzt genau ein Anwendungspaket* aus Tabelle 5.7 mit der Relation *besitzt Anwendungspaket.*

Tabelle 5.9 zeigt das Axiom *Java Paket Dateiname*, das festlegt, welche Endung der *Dateiname* einer Instanz der Klasse *Java Paket* besitzen muss. Ebenso könnte die Definition eines Axioms für die Klasse *Deployment Deskriptor* aussehen für die Endung `xml`, mit dem Zusatz, dass diese *Datei* in einem *Verzeichnis* mit dem *Dateiname* `META-INF` oder `WEB-INF` liegen muss.

Tabelle 5.9: Auszug aus der Tabelle der formalen Axiome der entwickelten Ontologie für das Axiom *Java Paket Dateiname*.

Name des Axioms	Java Paket Dateiname
Beschreibung	Der Dateiname eines Java Pakets muss entweder mit `war`, `ear`, `rar` oder `jar` enden

Ausdruck	`forall(?X)`
	`  ([Java Paket](?X) and`
	`  ((?X)[Dateiname] matches "*." + JAR_FILENAME or`
	`  (?X)[Dateiname] matches "*." + WAR_FILENAME or`
	`  (?X)[Dateiname] matches "*." + EAR_FILENAME or`
	`  (?X)[Dateiname] matches "*." + RAR_FILENAME))`

Klassen	Java Paket
Attribute	Dateiname
Konstanten	JAR_FILENAME, WAR_FILENAME, EAR_FILENAME, RAR_FILENAME
Relationen	–
Variablen	?X

5.2.10 Definieren von Regeln

Ähnlich der Aufgabe zur Definition von Axiomen müssen die Regeln einer Ontologie identifiziert und definiert werden. Regeln werden formal durch Bedingungen nach dem Schema *if <Bedingungen> then <Konsequenz>* beschrieben. METHONTOLOGY schlägt in diesem Zusammenhang vor die folgenden Informationen für jede Regel zu sammeln und festzuhalten [GPCFL04]:

- Name der Regel
- Beschreibung der Regel in normaler Sprache
- logischer Ausdruck, der die Regel formal in Prädikatenlogik erster Stufe beschreibt
- die Klassen, Attribute und Relationen, auf die sich die Regel bezieht
- die verwendeten Variablen

In Tabelle 5.10 wird die Regel *Typ eines Java Pakets einer Webanwendung* definiert. Diese Regel sagt aus, dass der *Typ* des *Anwendungspakets* einer *Webanwendung* den Wert „war“ haben muss. Ähnliche Regeln gibt es für die Klassen *Enterprise Anwendung* und *Ressourcen Adapter* mit den *Typen* „ear“ und „rar“ und den Konstanten EAR_FILENAME und RAR_FILENAME. Diese Regeln heißen *Typ eines Java Pakets einer Enterprise Anwendung* und *Typ eines Java Pakets eines Ressourcen Adapters* und sind analog zur Regel *Typ eines Java Pakets einer Webanwendung* definiert. Auf die Darstellung dieser beider Regeln wird verzichtet, da sie im Rahmen dieser Arbeit zu umfangreich wäre und keinen weiteren Mehrwert liefert.

Tabelle 5.10: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Typ eines Java Pakets einer Webanwendung*.

Name der Regel	Typ eines Java Pakets einer Webanwendung
Beschreibung	Der Typ des Java Pakets einer Webanwendung muss auf „war“ sein

Ausdruck	`if [Webanwendung](?X) and`
	`  [Anwendungspaket](?Y) and`
	`  [besitzt Anwendungspaket](?X, ?Y)`
	`then (?Y)[Typ] = WAR_FILENAME`

Klassen	Webanwendung, Anwendungspaket
Attribute	Typ
Konstanten	WAR_FILENAME
Relationen	besitzt Anwendungspaket
Variablen	?X, ?Y

Tabelle 5.11: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Webanwendung wegen Typ war*.

Name der Regel	Webanwendung wegen Typ war
Beschreibung	Ist der Typ eines Java Pakets „war“, dann ist die zugehörige Applikation eine Webanwendung

Ausdruck	`if [Applikation](?X) and`
	`  [Anwendungspaket](?Y) and`
	`  [besitzt Anwendungspaket](?X, ?Y) and`
	`  (?Y)[Typ] == WAR_FILENAME`
	`then [Webanwendung](?X)`

Klassen	Applikation, Anwendungspaket, Webanwendung
Attribute	Typ
Konstanten	WAR_FILENAME
Relationen	besitzt Anwendungspaket
Variablen	?X, ?Y

Die Umkehrung der Regel *Webanwendung wegen Typ war* aus Tabelle 5.10 ist in Tabelle 5.11 abgebildet und hat den Namen *Webanwendung wegen Typ war*. Diese Regel besagt, dass wenn der *Typ* eines *Anwendungspakets* einer *Applikation* den Wert „war“ besitzt, die *Applikation* eine *Webanwendung* ist. Das gilt natürlich ebenso für `ear` und `rar`-Dateien. Die korrespondierenden Regeln *Enterprise Anwendung wegen Typ ear* und *Ressourcen Adapter wegen Typ rar* entsprechen der Regel *Webanwendung wegen Typ war* mit den entsprechend anderen Klassen (*Enterprise Anwendung* und *Ressourcen Adapter*) und Konstanten (EAR_FILENAME und

RAR_FILENAME). Auf ihre Darstellung wird verzichtet, da sie im Rahmen dieser Arbeit zu umfangreich wäre und keinen Mehrwert liefert.

Tabelle 5.12: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Dateiendung war bedingt Typ des Java Pakets*.

Name der Regel	Dateiendung war bedingt Typ des Java Pakets
Beschreibung	Ist die Endung des Dateinamens eines Java Pakets `war`, so ist der Typ des Java Pakets „war“

Ausdruck	`if [Java Paket](?X) and`
	`  (?X)[Dateiname] matches "*." + WAR_FILENAME`
	`then (?X)[Typ] = WAR_FILENAME`

Klassen	Java Paket
Attribute	Dateiname, Typ
Konstanten	WAR_FILENAME
Relationen	–
Variablen	?X

Mit der in Tabelle 5.12 vorgestellten Regel *Dateiendung war bedingt Typ des Java Pakets* wird anhand des Attributes *Dateinamen* eines *Java Pakets* das Attribut *Typ* auf „war“ festgelegt, wenn *Dateiname* auf `war` endet. Für die Endungen `jar`, `ear` und `rar` existieren die Regeln *Dateiendung jar bedingt Typ eines Java Pakets*, *Dateiendung ear bedingt Typ eines Java Pakets* und *Dateiendung rar bedingt Typ eines Java Pakets*, die hier aus Platzgründen nicht aufgeführt sind, aber analog zur Regel *Dateiendung war bedingt Typ des Java Pakets* definiert sind.

Tabelle 5.13: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Typ war eines Java Pakets erzwingt Anwendungspaket*.

Name der Regel	Typ war eines Java Pakets erzwingt Anwendungspaket
Beschreibung	Ist der Typ eines Java Pakets „war“, dann ist das Java Paket ein Anwendungspaket

Ausdruck	`if [Java Paket](?X) and`
	`  (?X)[Typ] == WAR_FILENAME`
	`then [Anwendungspaket](?X)`

Klassen	Java Paket, Anwendungspaket
Attribute	Typ
Konstanten	WAR_FILENAME
Relationen	–
Variablen	?X

Tabelle 5.13 legt die Regel *Typ war eines Java Pakets erzwingt Anwendungspakt* fest. Diese Regel definiert, dass wenn der *Typ* eines *Java Pakets* „war“ ist, dann ist diese Instanz auch ein *Anwendungspaket*, was wiederum mit dem Axiom *Anwendungspaket gehört zu genau einer Applikation* (Tabelle 5.8) bedeutet, dass eine *Applikation* dafür vorhanden sein muss. Die

Regeln *Typ ear eines Java Pakets erzwingt Anwendungspakt* und *Typ rar eines Java Pakets erzwingt Anwendungspakt* sind für die Typen „ear“ und „rar“ definiert. Die Beschreibung dieser beider Regeln würde den Rahmen dieser Arbeit sprengen und liefert keinen zusätzlichen Mehrwert.

Tabelle 5.14: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Datei ist direkt enthalten in*.

Name der Regel	Datei ist direkt enthalten in
Beschreibung	Eine Datei ist direkt enthalten in einer anderen Datei, wenn sie entweder Paketbestandteil oder Verzeichnisbestandteil dieser Datei ist

Ausdruck	`if [Datei](?X) and`
	`  ([Verzeichnis](?Y) and`
	`  [Verzeichnisbestandteil](?Y, ?X))`
	`  or`
	`  (([Java Paket](?Y) and`
	`  [Paketbestandteil](?Y, ?X))`
	`then [Datei direkt enthalten in](?X, ?Y)`

Klassen	Datei, Verzeichnis, Java Paket
Attribute	–
Konstanten	–
Relationen	Verzeichnisbestandteil, Paketbestandteil, Datei direkt enthalten in
Variablen	?X, ?Y

In Tabelle 5.14 wird die Definition der Regel *Datei ist direkt enthalten in* gegeben. Diese Regel besagt, dass eine Instanz der Klasse *Datei* über der Relation *Datei direkt enthalten in* mit einer anderen *Datei* verbunden ist, wenn die andere *Datei* ein *Verzeichnis* ist und die *Datei* als *Verzeichnisbestandteil* enthält oder ein *Java Paket* ist und die *Datei* als *Paketbestandteil* enthält. Dies entspricht dem normalen Verständnis von Dateien.
Tabelle 5.15 stellt eine erste Bedingung für die Relation *Datei enthalten in* dar. Die hier definierte Regel *Datei ist enthalten in wegen direkt enthalten* beschreibt, dass eine *Datei* in einer anderen *Datei* enthalten ist, wenn sie direkt in dieser *Datei* enthalten ist. Sie ist nach der Regel *Datei ist direkt enthalten in* also entweder *Paketbestandteil* oder *Verzeichnisbestandteil* der anderen *Datei*.
Tabelle 5.16 beschreibt die Regel *Datei ist enthalten in* für die Relation *Datei enthalten in* und erweitert die Regel *Datei ist enthalten in wegen direkt enthalten* aus Tabelle 5.15. Eine *Datei* ist in einer anderen *Datei* enthalten, wenn es eine weiter *Datei* gibt, die die erste *Datei* enthält und selbst in der zweiten *Datei* enthalten ist. Das spiegelt eine Struktur wieder, die unabhängig davon ist, ob eine Datei ein *Verzeichnis* oder ein *Java Paket* ist.
Die Regel *Applikation Java EE Eigenschaft entspricht* aus Tabelle 5.17 beschreibt, wie eine *Java EE Eigenschaft* von einer *Applikation* mit einer *Java EE Eigenschaft* eines *Anwen-*

Tabelle 5.15: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Datei ist enthalten in wegen direkt enthalten*.

Name der Regel	Datei ist enthalten in wegen direkt enthalten
Beschreibung	Eine Datei ist enthalten in einer anderen Datei, wenn sie direkt in dieser Datei enthalten ist

Ausdruck	`if [Datei](?X) and`
	`  [Datei](?Y) and`
	`  [Datei direkt enthalten in](?X, ?Y)`
	`then [Datei enthalten in](?X, ?Y)`

Klassen	Datei
Attribute	–
Konstanten	–
Relationen	Datei direkt enthalten in, Datei enthalten in
Variablen	?X, ?Y

Tabelle 5.16: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Datei ist enthalten in*.

Name der Regel	Datei ist enthalten in
Beschreibung	Eine Datei X ist enthalten in einer anderen Datei Y, wenn es eine weitere Datei Z gibt, die in Y enthalten ist und die X enthält

Ausdruck	`if [Datei](?X) and`
	`  [Datei](?Y) and`
	`  [Datei](?Z) and`
	`  [Datei enthalten in](?X, ?Z) and`
	`  [Datei enthalten in](?Z, ?Y)`
	`then [Datei enthalten in](?X, ?Y)`

Klassen	Datei
Attribute	–
Konstanten	–
Relationen	Datei enthalten in
Variablen	?X, ?Y, ?Z

dungsservers über die Relation *Applikation Eigenschaft entspricht* korrespondiert. Auf diese Art und Weise kann überprüft werden, welche Eigenschaften einer *Applikation* mit denen eines *Anwendungsservers* übereinstimmen. Falls die *Java EE Eigenschaft* bei einem *Anwendungsserver* nicht unterstützt wird, kann die *Applikation* auf diesem *Anwendungsserver* auch nicht funktionieren. Andernfalls müssen noch die vorhandenen *Java EE Attribute* und deren Werte auf Existenz überprüft werden.

Die Regel *Applikation Java EE Eigenschaft entspricht* bezieht sich ausschließlich auf *Java EE Eigenschaften*, deren Überelemente mit der Relation *Applikation Eigenschaft entspricht* verbunden sind. Diese Regel greift also nur bei *Java EE Eigenschaften*, die auch Überelemente besitzen. *Java EE Eigenschaften*, die keine Überelemente besitzen, wie beispielsweise `<web>`

Tabelle 5.17: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Applikation Java EE Eigenschaft entspricht*.

Name der Regel	Applikation Java EE Eigenschaft entspricht
Beschreibung	Eine Java EE Eigenschaft einer Applikation entspricht einer Java EE Eigenschaft eines Anwendungsservers, wenn die jeweils übergeordneten Java EE Eigenschaften (falls vorhanden) sich auch entsprechen und die Namen der Java EE Eigenschaft gleich sind

Ausdruck	`if [Java EE Eigenschaft](?X) and`
	`[Applikation](?A) and`
	`[verwendet Eigenschaft](?A, ?X) and`
	`[Java EE Eigenschaft](?Y) and`
	`[Anwendungsserver](?B) and`
	`[unterstützt Eigenschaft](?B, ?Y) and`
	`(?X)[Name] == (?Y)[Name] and`
	`[Java EE Eigenschaft](?XX) and`
	`[Teileigenschaft](?XX, ?X) and`
	`[verwendet Eigenschaft](?A, ?XX) and`
	`[Java EE Eigenschaft](?YY) and`
	`[Teileigenschaft](?YY, ?Y) and`
	`[unterstützt Eigenschaft](?B, ?YY) and`
	`[Applikation Eigenschaft entspricht](?XX, ?YY)`
	`then [Applikation Eigenschaft entspricht](?X, ?Y)`

Klassen	Java EE Eigenschaft, Applikation, Anwendungsserver
Attribute	Name
Konstanten	–
Relationen	verwendet Eigenschaft, unterstützt Eigenschaft, Teileigenschaft, Applikation Eigenschaft entspricht
Variablen	?A, ?B, ?X, ?XX, ?Y, ?YY

im Deployment Deskriptor `web.xml`, werden von dieser Regel nicht berührt. Für diese gibt es die zusätzliche Regel *Applikation Java EE Eigenschaft auf oberster Ebene entspricht* aus Tabelle 5.18.
Mit der Regel *Kompatibilität eines Java EE Attributs* (Tabelle 5.19) wird beschrieben, ob und wann ein *Java EE Attribut* mit einem *Anwendungsserver* kompatibel ist. Das ist dann der Fall, wenn die *Java EE Eigenschaft*, deren Teil das *Java EE Attribut* ist, einer anderen *Java EE Eigenschaft* des *Anwendungsservers* entspricht und *Attributsname* und *Attributswert* beim *Anwendungsserver* vorhanden sind. Hiermit kann auf einfache Weise geprüft werden, ob eine vorhandene *Applikation* bezüglich der *Java EE Attribute* auf einem *Anwendungsserver*

Tabelle 5.18: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Applikation Java EE Eigenschaft auf oberster Ebene entspricht.*

Name der Regel	Applikation Java EE Eigenschaft auf oberster Ebene entspricht
Beschreibung	Eine Java EE Eigenschaft auf höchster Ebene einer Applikation entspricht einer Java EE Eigenschaft eines Anwendungsservers, wenn die Namen der Java EE Eigenschaften gleich sind

Ausdruck	`if [Java EE Eigenschaft](?X) and`
	`  [Applikation](?A) and`
	`  [verwendet Eigenschaft](?A, ?X) and`
	`  [Java EE Eigenschaft](?Y) and`
	`  [Anwendungsserver](?B) and`
	`  [unterstützt Eigenschaft](?B, ?Y) and`
	`  (?X)[Name] == (?Y)[Name] and`
	`  NOT exists ([Java EE Eigenschaft](?XX) with`
	`  [Teileigenschaft](?XX, ?X)) and`
	`  NOT exists ([Java EE Eigenschaft](?YY) with`
	`  [Teileigenschaft](?YY, ?Y))`
	`then [Applikation Eigenschaft entspricht](?X, ?Y)`

Klassen	Java EE Eigenschaft, Applikation, Anwendungsserver
Attribute	Name
Konstanten	–
Relationen	verwendet Eigenschaft, unterstützt Eigenschaft, Teileigenschaft, Applikation Eigenschaft entspricht
Variablen	?A, ?B, ?X, ?XX, ?Y, ?YY

lauffähig ist.

5.2.11 Definieren von Instanzen

Mit der Definition der Regeln ist das konzeptionelle Modell der Ontologie fertig erstellt. Anschließend können noch Instanzen hinzugefügt werden, wenn dies nötig ist. Hier ist für jede Instanz der Instanzname, die zugehörige Klasse und die entsprechenden Attribute bestehend aus Attributnamen und Attributwerten anzugeben.
In Tabelle 5.20 werden ein paar Instanzen der entwickelten Ontologie vorgestellt. Das sind ausschließlich Instanzen der Klasse *Anwendungsserver* und deren Eigenschaften. Das liegt daran, dass nur Applikationen automatisiert analysiert werden sollen. Eine gleichzeitige Analyse von Anwendungsserver und Applikation ist nicht vorgesehen und realisierbar. Aus diesem Grund müssen die Eigenschaften und Parameter von Anwendungsservern schon vorher defi-

Tabelle 5.19: Auszug aus der Regeltabelle der entwickelten Ontologie für die Regel *Kompatibilität eines Java EE Attributs*.

Name der Regel	Kompatibilität eines Java EE Attributs
Beschreibung	Ein Java EE Attribut ist dann mit einem Anwendungsserver kompatibel, wenn die korrespondierende Java EE Eigenschaft des Attributs auch auf dem Anwendungsserver vorhanden ist und das Attribut selbst mit Name und Wert verwendet werden darf

Ausdruck	`if [Java EE Attribut](?X) and`
	`[Java EE Eigenschaft](?XE) and`
	`[besitzt Java EE Attribut](?XE, ?X) and`
	`[Java EE Eigenschaft](?YE) and`
	`[Anwendungsserver](?Z) and`
	`[unterstützt Eigenschaft](?Z, ?YE) and`
	`[Applikation Eigenschaft entspricht](?XE, ?YE) and`
	`[Java EE Attribut](?Y) and`
	`[besitzt Java EE Attribut](?YE, ?Y) and`
	`(?X)[Attributsname] == (?Y)[Attributsname] and`
	`(?X)[Attributswert] matches (?Y)[Attributswert]`
	`then [Java EE Attribut kompatibel mit](?X, ?Z)`

Klassen	Java EE Attribut, Java EE Eigenschaft, Anwendungsserver
Attribute	Attributsname, Attributswert
Konstanten	–
Relationen	besitzt Java EE Attribut, Java EE Attribut kompatibel mit
Variablen	?X, ?Y, ?XE, ?YE, ?Z

niert und in der Ontologie hinterlegt worden sein.

5.3 Formale Kompetenzfragen

In diesem Kapitel werden die in Kapitel 5.1 beschriebenen Kompetenzfragen formal definiert. Auf diese Art können sie mit wenig Übersetzungsaufwand für die entwickelte Ontologie verwendet werden. Die Schreibweise einer formalen Kompetenzfrage orientiert sich an folgendem Schema:

```
FIND
    <Zielvariablen>
GIVEN
    <Variablen>
SUBJECT TO
    <Bedingungen>
```

Tabelle 5.20: Auszug aus der Instanztabelle der entwickelten Ontologie.

Instanzname	**Klasse**	**Attribut**	**Wert**
Oracle GlassFish v3.1.2.2 Full	Anwendungs-server	Servername	Oracle GlassFish
		Version	3.1.2.2
		Ausführung	Full Plattform
IBM WebSphere v8.5.5 Liberty	Anwendungs-server	Servername	IBM WebSphere
		Version	8.5.5
		Ausführung	Liberty
RedHat WildFly v9.0.1 Full	Anwendungs-server	Servername	RedHat WildFly
		Version	9.0.1
		Ausführung	Full
Oracle WebLogic v12.1.3	Anwendungs-server	Servername	Oracle WebLogic
		Version	12.1.3
		Ausführung	Standard (SERVER_ STANDARD_REALISA TION)
Apache Tomcat v8.0.24	Anwendungs-server	Servername	Apache Tomcat
		Version	8.0.24
		Ausführung	Standard (SERVER_ STANDARD_REALISA TION)

Hierbei sind die `<Zielvariablen>` die gesuchten Variablen und die `<Variablen>` die vorgegebenen Variablen unter den Bedingungen `<Bedingungen>`. Es werden alle Instanzen gefunden, die den Bedingungen entsprechen. Die folgende Aufzählung beschreibt die Kompetenzfragen auf diese Art und Weise:

1. **Welche Eigenschaften besitzt die analysierte Anwendung?**

```
FIND
    [Java EE Eigenschaft](?X)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    ([verwendet Eigenschaft](?Y, ?X)) or
        ([Anwendungspaket](?Z) and
        [besitzt Anwendungspaket](?Y, ?Z) and
        [Anwendungspaket](?A) and
        [Datei enthalten in](?A, ?Z) and
        [Applikation](?B) and
        [ist Anwendungspaket von](?A, ?B) and
        [verwendet Eigenschaft](?B, ?X))
```

   Mit dieser Frage werden alle *Java EE Eigenschaften* einer *Applikation* und aller darin enthaltener weiterer *Applikationen* abgerufen.

2. Wie sieht die Ordner- und Dateistruktur der analysierten Anwendung aus?

```
FIND
    [Datei](?X)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    [Anwendungspaket](?Z) and
    [besitzt Anwendungspaket](?Y, ?Z) and
    [Datei enthalten in](?X, ?Z)
```

Mit der Antwort auf diese Frage erhält man die Ordner und Paketstruktur der Anwendung. Hier wird nicht zwischen einem *Verzeichnis* oder *Java Paket* unterschieden.

3. Welche Art von Anwendung ist die analysierte Applikation?

```
FIND
    [Typ](?X)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    [Anwendungspaket](?X) and
    [besitzt Anwendungspaket](?Y, ?X)
```

Für diese Frage muss nur das Attribut *Typ* des zur *Applikation* gehörenden *Anwendungspakets* abgefragt werden.

4. Aus welchen Arten von Anwendungen besteht die analysierte Applikation?

```
FIND
    [Typ](?X)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    [Anwendungspaket](?X) and
    [Anwendungspaket](?Z) and
    [Datei enthalten in](?X, ?Z) and
    [besitzt Anwendungspaket](?Y, ?Z)
```

Mit dieser Frage werden alle in einer *Applikation* enthaltenen anderen *Applikationen* abgefragt und deren Attribut *Typ* ermittelt.

5. Welche Java EE APIs verwendet die Anwendung?

Diese Frage ist nicht so einfach zu beantworten. Im Grunde ist hierfür der Classpath der Anwendung zu überprüfen und welche Arten von Klassen verwendet werden. Hierfür muss natürlich jede Klasse einer API zugeordnet werden. Sobald eine dieser Klassen auf dem Classpath der Anwendung liegt, wird diese API verwendet.

6. Welche Eigenschaften einer Anwendung sind mit einem speziellen Server kompatibel?

```
FIND
    [Java EE Eigenschaft](?X)
GIVEN
    [Applikation](?Y), [Anwendungsserver](?Z)
SUBJECT TO
    [verwendet Eigenschaft](?Y, ?X) and
    [Java EE Eigenschaft](?E) and
    [Applikation Eigenschaft entspricht](?X, ?E) and
    [unterstützt Eigenschaft](?Z, ?E) and
    (ALL ([Java EE Attribut](?A) and
        [besitzt Java EE Attribut](?X, ?A))
        FULLFILL [Java EE Attribut kompatibel mit](?A, ?Z))
```

Mit dieser Frage werden alle *Java EE Eigenschaften* einer *Applikation* ausgegeben, die von einem *Anwendungsserver* unterstützt werden und deren *Java EE Attribute* kompatibel mit *Anwendungsserver* sind. Man könnte meinen, dass es mit der Regel *Kompatibilität eines Java EE Attributs* (Tabelle 5.19) ausreichen würde, nur die enthaltenen *Java EE Attribute* auf Kompatibilität zu überprüfen. Das funktioniert allerdings nur, wenn wirklich *Java EE Attribute* in der *Java EE Eigenschaft* enthalten sind.

7. Welche Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?

```
FIND
    [Java EE Eigenschaft](?X)
GIVEN
    [Applikation](?Y), [Anwendungsserver](?Z)
SUBJECT TO
    [verwendet Eigenschaft](?Y, ?X) and
    (([Java EE Eigenschaft](?E) and
        NOT [Applikation Eigenschaft entspricht](?X, ?E)) or
    ([Java EE Eigenschaft](?E) and
        [Applikation Eigenschaft entspricht](?X, ?E) and
        NOT [unterstützt Eigenschaft](?Z, ?E)) or
    (NOT ALL ([Java EE Attribut](?A) and
        [besitzt Java EE Attribut](?X, ?A))
        FULLFILL [Java EE Attribut kompatibel mit](?A, ?Z)))
```

Eine *Java EE Eigenschaft*, die nicht mit einem *Anwendungsserver* kompatibel ist, entspricht entweder keiner *Java EE Eigenschaft* des *Anwendungsservers*, wird so nicht vom *Anwendungsserver* unterstützt oder enthält mindestens ein *Java EE Attribut*, das nicht mit dem *Anwendungsserver* kompatibel ist.

8. Wie viele Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?

Die Antwort auf diese Frage ist die Anzahl der Elemente, die in der vorherigen Frage gefunden wurden.

9. Kann die Anwendung auf einem speziellen Server funktionieren?

Die vorherige Frage kann als erster Ansatzpunkt für diese Frage herangezogen werden. Wenn die Anzahl der Elemente, die nicht mit dem speziellen Server kompatibel sind, größer null ist, dann kann die Anwendung nicht kompatibel sein und es muss nichts weiter betrachtet werden. Andernfalls müssen noch andere Aspekte der Anwendung überprüft werden, wie die Elemente auf dem Class Path beispielsweise:

```
FIND
    [Java Klasse](?X)
GIVEN
    [Applikation](?Y), [Anwendungsserver](?Z)
SUBJECT TO
    [Java Klasse](?A) and
    [Class Path Element](?Y, ?A) and
    [benötigt Klasse](?A, ?X) and
    NOT ([Class Path Element](?Y, ?X) and
        [Server Class Path Element](?Z, ?X))
```

Wenn man die Anzahl aller Elemente (Eigenschaften, Class Path Elemente, usw.), die nicht mit einem Anwendungsserver kompatibel sind, mit *Anzahl nicht kompatibler Elemente* eines *Anwendungsservers* für eine *Applikation* bezeichnen würde, dann müsste gelten, damit eine *Applikation* $?Y$ auf einem *Anwendungsserver* $?Z$ lauffähig ist:
`[Anzahl nicht kompatibler Elemente](?Z, ?Y) = 0`

10. Auf welchen Anwendungsservern ist die analysierte Anwendung lauffähig?

Diese Frage ist mit dem in der vorigen Frage definierten Attribut *Anzahl nicht kompatibler Elemente* zu beantworten:

```
FIND
    [Anwendungsserver](?Z)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    [Anzahl nicht kompatibler Elemente](?Z, ?Y) == 0
```

Die *Anzahl nicht kompatibler Elemente* ist eher eine Relation zwischen einem *Anwendungsserver* und einer *Applikation* mit einem numerischen Wert. Diese Art von Objekt ist in der Ontologie nicht direkt abbildbar, muss also unabhängig davon berechnet werden.

11. Welcher Server wäre der geeignetste für die analysierte Anwendung?

Mit dieser Frage soll nicht der bestmögliche Anwendungsserver gefunden werden, auf dem die Anwendung lauffähig ist, sondern der Anwendungsserver, für den die geringsten Anpassungen an der Anwendung gemacht werden müssen:

```
FIND
    [Anwendungsserver](?Z)
GIVEN
    [Applikation](?Y)
SUBJECT TO
    ?Z = [Anwendungsserver](?X) with
         min([Anzahl nicht kompatibler Elemente](?X, ?Y))
```

5.4 Zusammenfassung

In diesem Kapitel wurde die Konzeptualisierung der Ontologie durchgeführt. Hierzu gehörten neben den Schritten aus METHONTOLOGY auch das Definieren von formalen Kompetenzfragen. Der nächste Schritt in der Entwicklung der Ontologie ist die Implementierung in einer formalen Ontologiesprache. Dies wird in Kapitel 6.1 für die Ontologiesprache OWL durchgeführt. Anschließend wird in Kapitel 6.2 die formale Ontologie anhand der hier entwickelten Kompetenzfragen evaluiert.

6 Implementierung der Ontologie in maschinenlesbarer Sprache

Dieses Kapitel beschäftigt sich mit der Implementierung der in Kapitel 5 entwickelten Ontologie. Hierfür wird in Kapitel 6.1 die Ontologie mit der Ontologiesprache owl implementiert. Anschließend wird der geforderte Umfang mit den definierten Kompetenzfragen in Kapitel 6.2 überprüft. Diese Evaluation stellt eine Korrektheitsprüfung der Implementierung aus Kapitel 6.1 dar.

6.1 Entwicklung der Ontologie

In diesem Kapitel werden die in der Konzeptionsphase der Ontologie beschriebenen Klassen, Attribute, Relationen, Axiome und Regeln mittels eines Ontologieeditors in eine Modellierungssprache für eine Ontologie übersetzt.
Es existieren sehr viele verschiedene Ontologieeditoren mit sehr unterschiedlichen Umfängen. Die erste Wahl für einen solchen Editor ist *Protégé*, das von der Stanford University entwickelt wird und kostenlos zur Verfügung steht [MB12]. Nach [GDD09] ist Protégé das führende Programm zur Entwicklung und Modellierung einer Ontologie und wird in über 60% aller Ontologieprojekte eingesetzt. Das liegt unter anderem an der umfangreichen Ausstattung von Protégé, das OWL und RDFS komplett unterstützt. Die für diese Sprachen gebräuchliche Erweiterung für Regeln, die *Semantic Web Rule Language* (kurz SWRL) [HPSB+04], ist auch von Protégé unterstützt, was für die hier entwickelte Ontologie von großer Bedeutung für Axiome und Regeln ist. Für diese Arbeit wird mit der neuesten Version 5.0 Beta 17 gearbeitet, da mit der aktuell stabilen Version 4.3 keine Möglichkeit besteht SWRL-Regeln komfortabel hinzuzufügen.
Um die durch Regeln definierten Schlussfolgerungen ziehen zu können, wurde der „Reasoner“ *Pellet* verwendet. Die Kompetenzfragen wurden mittels *DL-Query* realisiert.
Die in diesem Kapitel beispielhaft aufgeführten Elemente der Ontologie dienen der Präsentation der Ergebnisse, sind allerdings nicht vollständig. Die komplette Ontologie ist als Beilage zu dieser Arbeit in der Datei `javaeemigration.owl` zu finden. In den nachfolgenden Unterkapiteln wird auf die Definition von einzelnen Elementen wie Klassen, Relationen und Regeln eingegangen.

6.1.1 Klassendefinitionen

In diesem Kapitel wird auf die Definition von Klassen und deren Instanzattribute eingegangen. Die Ontologie wurde mittels OWL und RDFS im entsprechenden XML-Schema festgelegt. Alle hier gezeigten Beispiele sind direkt im äußersten Element `<Ontology>` enthalten. Es wird hier beispielhaft die Klasse *Attribut* herangezogen, die mittels OWL folgendermaßen definiert ist:

```
<Declaration>
    <Class IRI="#Attribut"/>
</Declaration>
<Declaration>
    <Class IRI="#Java_EE_Attribut"/>
</Declaration>
<Declaration>
    <Class IRI="#Funktionsattribut"/>
</Declaration>

<SubClassOf>
    <Class IRI="#Java_EE_Attribut"/>
    <Class IRI="#Attribut"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Funktionsattribut"/>
    <Class IRI="#Attribut"/>
</SubClassOf>
```

Mit dem Tag `Declaration` werden alle Objekte in der Ontologie erwähnt und einem Typ, in diesem Fall `Class` zugeordnet. Mit `SubClassOf` wird beschrieben, dass die erste `Class` eine Subklasse der zweiten `class` ist. Hier werden also die Klassen `Attribut`, `Java_EE_Attribut` und `Funktionsattribut` analog zu den Klassen *Attribut*, *Java EE Attribut* und *Funktionsattribut* (vgl. Abschnitt 5.2.2) definiert.

6.1.2 Instanzattribute

Die Klasse *Attribut* besitzt nach Abschnitt 5.2.6 die beiden Instanzattribute *Attributsname* und *Attributswert*, die mittels OWL folgendermaßen definiert sind:

```
<Declaration>
    <DataProperty IRI="#Attributsname"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#Attributswert"/>
</Declaration>

<DataPropertyDomain>
    <DataProperty IRI="#Attributsname"/>
    <Class IRI="#Attribut"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#Attributswert"/>
    <Class IRI="#Attribut"/>
</DataPropertyDomain>

<DataPropertyRange>
```

```
    <DataProperty IRI="#Attributsname"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
<DataPropertyRange>
    <DataProperty IRI="#Attributswert"/>
    <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
```

Mit `Declaration` werden Instanzattribute als `DataProperty` eingeführt. Die zugehörige Klasse wird mit `DataPropertyDomain` zur entsprechenden `DataProperty` hinzugefügt. In diesem Zusammenhang macht es keinen Sinn ein Instanzattribut zu mehreren Klassen hinzuzufügen, weil Objekte mit diesem Instanzattribut dann zu jeder dieser Klassen gleichzeitig gehören könnten und dadurch zu einigen Problemen führen kann. Aus diesem Grund sind in der hier entwickelten Ontologie keine Instanzattribute mit gleichem Namen auf unterschiedlichen Klassen definiert. Mit `DataPropertyRange` kann für jedes Instanzattribut der Wertebereich festgelegt werden. Dieser Wertebereich ist entweder ein Typ, wie in diesem Fall `xsd:string` also String, kann aber auch einen tatsächlichen Wertebereich darstellen.
Hier sind die Instanzattribute `Attributsname` und `Attributswert` der Klasse `Attribut` zugeordnet und haben beide einen Wert vom Typ `xsd:string`.

6.1.3 Relationsdefinitionen

Die in Abschnitt 5.2.5 definierten Relationen werden in diesem Kapitel beispielhaft für die Klasse *Java EE Attribut* mit der Ontologiesprache OWL aufgezeigt. Die relevanten Relationen sind *ist Java EE Attribut von*, *Java EE Attribut kompatibel mit* und die zugehörigen Gegenrelationen *besitzt Java EE Attribut* bzw. *unterstützt Java EE Attribut*.

```
<Declaration>
    <ObjectProperty IRI="#besitzt_Java_EE_Attribut"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#ist_Java_EE_Attribut_von"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#Java_EE_Attribut_kompatibel_mit"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#unterstützt_Java_EE_Attribut"/>
</Declaration>

<InverseObjectProperties>
    <ObjectProperty IRI="#besitzt_Java_EE_Attribut"/>
    <ObjectProperty IRI="#ist_Java_EE_Attribut_von"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#Java_EE_Attribut_kompatibel_mit"/>
    <ObjectProperty IRI="#unterstützt_Java_EE_Attribut"/>
</InverseObjectProperties>
```

Die Relationen werden als `ObjectProperty` eingeführt und mit dem Tag `InverseObjectProperties` als Gegenrelationen definiert. So ist hier zu sehen, dass die Relationen `besitzt_Java_EE_Attribut` und `ist_Java_EE_Attribut_von` ebenso wie `Java_EE_Attribut_kompatibel_mit` und `unterstützt_Java_EE_Attribut` Gegenrelationen sind.

```
<ObjectPropertyDomain>
    <ObjectProperty IRI="#besitzt_Java_EE_Attribut"/>
    <Class IRI="#Java_EE_Eigenschaft"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#Java_EE_Attribut_kompatibel_mit"/>
    <Class IRI="#Java_EE_Attribut"/>
</ObjectPropertyDomain>

<ObjectPropertyRange>
    <ObjectProperty IRI="#besitzt_Java_EE_Attribut"/>
    <Class IRI="#Java_EE_Attribut"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#Java_EE_Attribut_kompatibel_mit"/>
    <Class IRI="#Anwendungsserver"/>
</ObjectPropertyRange>
```

Über die Tags `ObjectPropertyDomain` und `ObjectPropertyRange` werden die Ausgangs- und Zielklasse einer Relation definiert. Für jedes Relation/Gegenrelation-Paar ist es ausreichend, nur für eine der beiden Relationen die Ausgangs- und Zielklasse zu definieren, da die entsprechende Gegenrelation die Klassen in vertauschter Form als Ausgangs- und Zielklasse besitzt. In diesem Fall ist die Ausgangsklasse von `besitzt_Java_EE_Attribut` die Klasse `Java_EE_Eigenschaft` und von `Java_EE_Attribut_kompatibel_mit` die Klasse `Java_EE_Attribut`. Die Zielklasse dieser Relationen sind die Klassen `Java_EE_Attribut` bzw. `Anwendungsserver`.

```
<FunctionalObjectProperty>
    <ObjectProperty IRI="#ist_Java_EE_Attribut_von"/>
</FunctionalObjectProperty>
```

Mit dem Tag `FunctionalObjectProperty` wird festgelegt, dass jeder Instanz der Ausgangsklasse maximal eine Instanz der Zielklasse über diese Relation zugeordnet werden darf. Diese Eigenschaft nennt sich Funktionalität, also das Verhalten einer mathematischen Funktion, wo auch jedem Ausgangswert maximal ein Zielwert zugeordnet wird. In diesem Fall ist die Relation `ist_Java_EE_Attribut_von` funktional, was bedeutet, dass jede Instanz der Klasse `Java_EE_Attribut` maximal einer Instanz der Klasse `Java_EE_Eigenschaft` zugeordnet werden darf. Das entspricht den Definitionen aus Abschnitt 5.2.5.

6.1.4 Axiome

In OWL und Protégé ist es nicht möglich eigene Axiome exakt zu definieren. Die einzige Möglichkeit ist die Herangehensweise über Regeln. Regeln werden mit der *Semantic Web Rule Language* (kurz SWRL) definiert, die dann in OWL umgesetzt wird.

In diesem Kapitel wird beispielhaft das Axiom *Applikation besitzt genau ein Anwendungspaket* (vgl. Tabelle 5.8) in OWL-Notation ausgeführt. Dieses Axiom beschreibt, dass eine Instanz der Klasse *Applikation* immer genau eine Relation *besitzt Anwendungspaket* zu einer Instanz der Klasse *Anwendungspaket* besitzen muss. Es gibt eine einfache Möglichkeit, dieses Axiom als Regel in der Notation aus Abschnitt 5.2.10 umzuschreiben:

```
if [Applikation](?X)
then [Anwendungspaket](?Y) and
    [besitzt Anwendungspaket](?X, ?Y)
```

Diese neue Regel lässt sich dann einfach als SWRL-Regel mit OWL definieren und hat dann diese Form:

```
Rule: Applikation(?x) -> besitzt_Anwendungspaket(?x, ?y)
```

Das besagt, dass eine Instanz der Klasse `Applikation` immer eine Relation `besitzt_Anwendungspaket` besitzen muss. In OWL-Notation sieht diese neue Regel dann folgendermaßen aus:

```
<DLSafeRule>
    <Body>
        <ClassAtom>
            <Class IRI="#Applikation"/>
            <Variable IRI="urn:swrl#x"/>
        </ClassAtom>
    </Body>
    <Head>
        <ObjectPropertyAtom>
            <ObjectProperty IRI="#besitzt_Anwendungspaket"/>
            <Variable IRI="urn:swrl#x"/>
            <Variable IRI="urn:swrl#y"/>
        </ObjectPropertyAtom>
    </Head>
</DLSafeRule>
```

In dieser Notation beschreibt jedes Element `DLSafeRule` eine eigene Regel. `Body` stellt die Bedingung der Regel dar und `Head` die Folgerung. In diesem Fall besteht die Bedingung aus einem einzelnen Atom, nämlich einem `ClassAtom` für die Klasse `Applikation` und deren Variablenwert `?x` (in dieser Notation `urn:swrl#x`). Die Folgerung besteht aus einem `ObjectPropertyAtom`, also einem Relationsatom, mit der Relation `besitzt_Anwendungspaket` ausgehend von `?x` an eine Variable `?y`. Mit der Definition der Relation `besitzt_Anwendungspaket` muss `?y` eine Instanz der Klasse `Anwendungspaket` sein. Diese Zuweisung ist also überflüssig. Probleme gibt es in diesem Zusammenhang mit oder-Verknüpfungen. Es gibt keine Möglichkeit SWRL-Regeln mit oder-Verknüpfung aufzustellen. Protégé hat immer einen Syntaxfehler für diese Regeln angezeigt. Deshalb war es nicht möglich das Axiom *Java Paket Dateiname* in die Ontologie zu integrieren. Da diese Regel aber auch keine weitere Hilfe für die Auswertung der Ontologie darstellt, ist das von geringer Bedeutung für die Entwicklung und Evaluierung der Ontologie.

6.1.5 Regeln

Dieses Kapitel beschäftigt sich mit der Repräsentation von Regeln in der Ontologie. Die in Abschnitt 5.2.10 beschriebenen Regeln wurden soweit möglich mittels SWRL-Regeln in die OWL-Notation übertragen. Wie im vorherigen Kapitel dargestellt muss auch hier darauf hingewiesen werden, dass oder-Verknüpfungen in SWRL-Regeln von Protégé als Syntaxfehler identifiziert wurden und deshalb nicht verwendet werden konnten. Dieses Problem ließ sich bei oder-Verknüpfungen in der Bedingung aber leicht dadurch umgehen, dass aus einer Regel mehrere mit den einzelnen oder-Teilen erzeugt wurden.
Beispielhaft wird in diesem Kapitel auf die Regel *Datei ist direkt enthalten in* (vgl. Tabelle 5.14) eingegangen. Diese Regel besagt, dass eine Instanz A der Klasse *Datei*, direkt in einer anderen Instanz B enthalten ist, wenn B entweder eine Instanz der Klasse *Verzeichnis* ist und die Relation *Verzeichnisbestandteil* zwischen B und A vorhanden ist oder B ein *Java Paket* ist und die Relation *Paketbestandteil* vorhanden ist. Da hier eine oder-Verknüpfung in der Bedingung vorhanden ist, war diese Regel auf zwei neue Regeln aufzuteilen (hier in SWRL-Notation):

```
Rule: Datei(?x), Verzeichnis(?y), ist_Verzeichnisbestandteil_von(?x, ?y)
    -> Datei_direkt_enthalten_in(?x, ?y)
Rule: Datei(?x), Java_Paket(?y), ist_Paketbestandteil_von(?x, ?y)
    -> Datei_direkt_enthalten_in(?x, ?y)
```

In OWL Notation ist die Regel für die Klasse `Verzeichnis` folgendermaßen definiert:

```
<DLSafeRule>
    <Body>
        <ClassAtom>
            <Class IRI="#Datei"/>
            <Variable IRI="urn:swrl#x"/>
        </ClassAtom>
        <ClassAtom>
            <Class IRI="#Verzeichnis"/>
            <Variable IRI="urn:swrl#y"/>
        </ClassAtom>
        <ObjectPropertyAtom>
            <ObjectProperty IRI="#ist_Verzeichnisbestandteil_von"/>
            <Variable IRI="urn:swrl#x"/>
            <Variable IRI="urn:swrl#y"/>
        </ObjectPropertyAtom>
    </Body>
    <Head>
        <ObjectPropertyAtom>
            <ObjectProperty IRI="#Datei_direkt_enthalten_in"/>
            <Variable IRI="urn:swrl#x"/>
            <Variable IRI="urn:swrl#y"/>
        </ObjectPropertyAtom>
    </Head>
</DLSafeRule>
```

Wie bereits in Abschnitt 6.1.4 erwähnt beschreibt `DLSafeRule` eine Regel in der Ontologie, deren Bedingung im Bereich `body` und Folgerung im Bereich `Head` aufgelistet ist. Einzelne Bedingungen sind als einzelne Atome beschrieben, die unterschiedliche Typen abhängig von der Objektart besitzen. In diesem Fall existieren beispielsweise `ClassAtom` und `ObjectPropertyAtom`, also Klassenatome und Relationsatome. Da die Relation `ist_Verzeichnisbestandteil_von` voraussetzt, von einer Instanz der Klasse `Datei` zu einer Instanz der Klasse `Verzeichnis` zu zeigen, sind die beiden Klassenatome eigentlich überflüssig. Es wird hier jedoch aus Gründen der Vollständigkeit darauf hingewiesen.
SWRL erweitert OWL um einige BuiltIns (`BuiltInAtom`), also hinzugefügte Funktionen, wie beispielsweise `endsWith`. Es handelt sich dabei um eine Prüfung, ob ein String-Wert mit einem bestimmten Wert endet. Diese Funktionen sind für manche Regeln wie *Dateiendung war bedingt Typ des Java Pakets* (vgl. Tabelle 5.12) relevant, um bei Instanzattributen den Inhalt zu überprüfen.

6.1.6 Kompetenzfragen

Kompetenzfragen sind im Wesentlichen Abfragen an die Ontologie, die beliebig komplex werden können. Die einfachste Art zur Definition einer Abfrage in Protégé ist eine *DL-Query*. Die als DL-Query definierten Kompetenzfragen können in der Ontologie als `EquivalentClasses` gespeichert und können direkt ausgewertet werden.
Die erste Kompetenzfrage sucht alle Eigenschaften einer analysierten Anwendung und ihrer Teilanwendungen. Diese Abfrage lässt sich in der DL-Query-Notation für eine Instanz „TestApp“ der Klasse *Applikation* folgendermaßen beschreiben:

```
(ist_verwendete_Eigenschaft_von value TestApp) or
(ist_verwendete_Eigenschaft_von some
    (besitzt_Anwendungspaket some
        (Datei_enthalten_in some
            (ist_Anwendungspaket_von value TestApp))))
```

Diese Aussage beschreibt, dass alle Instanzen gesucht werden, die über der Relation `ist_verwendete_Eigenschaft_von` mit der Instanz „TestApp“ verbunden sind. Zusätzlich werden alle Instanzen gefunden, die über die Relation `ist_verwendete_Eigenschaft_von` mit Instanzen verbunden sind, die über die Relation `besitzt_Anwendungspaket` mit anderen Instanzen verbunden sind, die über die Relation `Datei_enthalten_in` mit wieder anderen Instanzen verbunden sind, die über die Relation `ist_Anwendungspaket_von` mit der Instanz „TestApp“ verknüpft sind.
In dieser Notation wird mit dem Wort `value` ein fester Wert bezeichnet und mit `some` eine einfache Verbindung dargestellt. Zusätzlich existieren noch die Schlüsselworte `only` für einen einzigen Wert, `min x` für mindestens x Verbindungen (optional auch ohne weiteren Werte) und `max x` für maximal x solche Verbindungen.
Das folgende Beispiel zeigt die obige mittels der DL-Query-Notation beschriebenen Kompetenzfrage *Welche Eigenschaften besitzt die analysierte Anwendung?* in OWL-Notation:

```
<EquivalentClasses>
    <Class IRI="#Welche_Eigenschaften_besitzt_die_Anwendung_TestApp"/>
    <ObjectUnionOf>
        <ObjectSomeValuesFrom>
```

```
                <ObjectProperty IRI="#ist_verwendete_Eigenschaft_von"/>
                <ObjectSomeValuesFrom>
                    <ObjectProperty IRI="#besitzt_Anwendungspaket"/>
                    <ObjectSomeValuesFrom>
                        <ObjectProperty IRI="#Datei_enthalten_in"/>
                        <ObjectHasValue>
                            <ObjectProperty IRI="#ist_Anwendungspaket_von"/>
                            <NamedIndividual IRI="#TestApp"/>
                        </ObjectHasValue>
                    </ObjectSomeValuesFrom>
                </ObjectSomeValuesFrom>
            </ObjectSomeValuesFrom>
            <ObjectHasValue>
                <ObjectProperty IRI="#ist_verwendete_Eigenschaft_von"/>
                <NamedIndividual IRI="#TestApp"/>
            </ObjectHasValue>
        </ObjectUnionOf>
    </EquivalentClasses>
```

Wie oben schon beschrieben, wird die Klasse `Welche_Eigenschaften_besitzt_die_Anwendung_TestApp` als `EquivalentClasses` definiert. Die `or`-Verknüpfung wird als `ObjectUnionOf` Element dargestellt. Die `some`-Verknüpfungen werden als `ObjectSomeValuesFrom` und die `value`-Verknüpfung als `ObjectHasValue` repräsentiert.
Aufgrund der generellen Struktur und Beschaffenheit von OWL-Ontologien sind Verknüpfungen mit `max x` oder `only` nicht sinnvoll verwendbar. Das liegt an der *Open-world Assumption* (OWA) von Ontologien. Das bedeutet, dass eine Behauptung solange als wahr gilt, solange sie wahr sein könnte. Das Gegenteil von OWA ist die *Closed-world Assumption* (CWA). Bei CWA gelten Behauptungen nur dann als wahr, wenn sie bewiesen sind. Am einfachsten lässt sich das an einem Beispiel erklären: [DS06]

- **Aussage**

  Christian kommt aus München

- **Frage**

  Kommt Anna aus München?

- **Closed-world Antwort**

  Nein

- **Open-world Antwort**

  Nicht definiert, könnte also wahr sein

Diese OWA bewirkt, dass mit den oben genannten Verknüpfungen keine Ergebnisse geliefert werden können, weil dieser Punkt nicht klar definiert sein kann, es könnte ja noch andere Elemente geben, auf die die Behauptung zutrifft, die aber noch nicht bekannt sind.
Die meisten Kompetenzfragen können als komplizierte DL-Queries geschrieben werden, liefern aber aufgrund der gerade erwähnten Probleme mit hoher Sicherheit keine Ergebnisse. Das

würde nur funktionieren, wenn die abgeleiteten zusammen mit den tatsächlich definierten Relationen in der Ontologie als für jedes Objekt abgeschlossen definiert würden. Dies ist für abgeleitete Relationen nicht möglich.

6.2 Evaluierung der Ontologie

In diesem Kapitel wird die in Kapitel 6.1 implementierte Ontologie mittels Kompetenzfragen maschinell evaluiert. Die Ontologie wird mit rein generischen Daten für Anwendungen und Anwendungsserver befüllt, damit die Kompetenzfragen überprüft werden können. Diese Evaluation stellt eine Korrektheitsprüfung der Implementierung aus Kapitel 6.1 dar.
In Abschnitt 6.2.1 werden die Instanzen beschrieben, die in der Ontologie definiert wurden und Abschnitt 6.2.2 beschäftigt sich mit den Ergebnissen der Kompetenzfragen. In Abschnitt 6.2.3 werden diese Ergebnisse zusammengefasst und die daraus resultierenden Konsequenzen dargelegt.

6.2.1 Testinstanzen

Die in der Ontologie definierten Instanzen werden in Tabelle 6.1 aufgelistet.

Tabelle 6.1: Instanzen in der Ontologie, die zum Evaluieren mittels der Kompetenzfragen verwendet werden.

Instanzname	**Klasse**	**Attribut/ Relation**	**Wert/nach**
TestServer	Anwendungs-server	Servername	„testname“
		Version	„1.0“
		Ausführung	„full“
TestEARApp	Applikation	Applikationsname	„test-ear“
		besitzt Anwendungspaket	TestEAR
TestWARApp	Applikation	Applikationsname	„test-war“
		besitzt Anwendungspaket	TestWAR
TestEARappxml	Deployment Deskriptor	Dateiname	„application.xml“
		ist Verzeichnisbe-standteil von	TestEARMETAINF
TestWARwebxml	Deployment Deskriptor	Dateiname	„web.xml“
		ist Verzeichnisbe-standteil von	TestWARWEBINF
TestWAR1	Java Paket	Dateiname	„test1.war“

Tabelle 6.1: Instanzen in der Ontologie, die zum Evaluieren mittels der Kompetenzfragen verwendet werden. – Fortsetzung

Instanzname	**Klasse**	**Attribut/ Relation**	**Wert/Zielinstanz**
TestWAR	Anwendungs-paket	Dateiname	„test.war“
		Typ	„war“
		Paketbestandteil	TestWARWEBINF
		ist Paketbestandteil von	TestEAR
TestEAR	Anwendungs-paket	Dateiname	„test.ear“
		Typ	„ear“
TestEARMETAINF	Verzeichnis	Dateiname	„META-INF“
		ist Paketbestandteil von	TestEAR
TestWARWEBINF	Verzeichnis	Dateiname	„WEB-INF“
TestWARclasses	Verzeichnis	Dateiname	„classes“
		ist Verzeichnisbe-standteil von	TestWARWEBINF
TestWARpackagedir	Verzeichnis	Dateiname	„testwar“
		ist Verzeichnisbe-standteil von	TestWARclasses
TestEARJEApp	Java EE Eigenschaft	Eigenschaftsname	„application“
		ist verwendete Eigenschaft von	TestEARApp
TestWARJEWeb	Java EE Eigenschaft	Eigenschaftsname	„web“
		ist verwendete Eigenschaft von	TestWARApp
TestServerJEApp	Java EE Eigenschaft	Eigenschaftsname	„application“
		ist unterstützte Eigenschaft von	TestServer
		Anwendungsserver Eigenschaft entspricht	TestEARJEApp
TestServerJEWeb	Java EE Eigenschaft	Eigenschaftsname	„web“
		ist unterstützte Eigenschaft von	TestServer
		Anwendungsserver Eigenschaft entspricht	TestWARJEWeb

Tabelle 6.1: Instanzen in der Ontologie, die zum Evaluieren mittels der Kompetenzfragen verwendet werden. – Fortsetzung

Instanzname	**Klasse**	**Attribut/ Relation**	**Wert/Zielinstanz**
TestEARJEAppAtt	Java EE Attribut	Attributsname	„Testboolean“
		Attributswert	„true“
		ist Java EE Attribut von	TestEARJEApp
TestWARJEWebAtt	Java EE Attribut	Attributsname	„Teststring“
		Attributswert	„blub“
		ist Java EE Attribut von	TestWARJEWeb
TestServerJEAppAtt	Java EE Attribut	Attributsname	„Testboolean“
		Attributswert	„true, false“
		ist Java EE Attribut von	TestServerJEApp
TestServerJEWebAtt	Java EE Attribut	Attributsname	„Teststring“
		Attributswert	„blub, blab, blib“
		ist Java EE Attribut von	TestServerJEWeb
TestWARKlasse	Java Klasse	Dateiname	„TestWARKlasse.class“
		Name	„TestWARKlasse“
		Package	„testwar“
		benötigt Klasse	TestAPIKlasse
		ist Class Path Element von	TestWARApp
TestAPIKlasse	Java Klasse	Dateiname	„TestAPIKlasse.class“
		Name	„TestAPIKlasse“
		Package	„testapi“
		ist Server Class Path Element von	TestServer

6.2.2 Testergebnisse

In diesem Kapitel werden die Kompetenzfragen an der entwickelten Ontologie zusammen mit den in Abschnitt 6.2.1 definierten Testinstanzen überprüft. Zu jeder Frage wird das erwartete Ergebnis und das tatsächliche Ergebnis angegeben, um die Funktionalität der Ontologie zu verifizieren. Die Fragen wurden immer für die Anwendung *TestEARApp* oder den Server *TestServer* gestellt.

1. **Welche Eigenschaften besitzt die analysierte Anwendung?**

   Erwartetes Ergebnis:

   TestEARJEApp, TestWARJEWeb

Geliefertes Ergebnis:

TestEARJEApp, TestWARJEWeb

Erläuterung:

Die Abfrage liefert genau die erwarteten Ergebnisse und verifiziert damit einen Teil der Ontologie.

2. **Wie sieht die Ordner- und Dateistruktur der analysierten Anwendung aus?**

Erwartetes Ergebnis:

TestEARMETAINF
TestEARappxml
TestWAR
TestWARWEBINF
TestWARclasses
TestWARPackage
TestWARKlasse
TestWARwebxml

Geliefertes Ergebnis:

TestEARMETAINF
TestEARappxml
TestWAR
TestWARWEBINF
TestWARclasses
TestWARPackage
TestWARKlasse
TestWARwebxml

Erläuterung:

Die Abfrage liefert genau die erwarteten Ergebnisse und verifiziert damit einen Teil der Ontologie.

3. **Welche Art von Anwendung ist die analysierte Applikation?**

Erwartetes Ergebnis:

ear

Geliefertes Ergebnis:

ear

Erläuterung:

Das Ergebnis der Abfrage ist genau wie erwartet. Es ist mit DL-Query nicht möglich Eigenschaftswerte von Instanzen als Abfrageergebnis zu erhalten. In den Abfragen können Eigenschaftswerte eingesetzt werden. Aus diesem Grund muss

diese Frage in vier unterschiedliche Fragen aufgespalten werden, wobei jede nach Instanzen eines bestimmten Typs (`jar`, `war`, `ear` und `rar`) sucht. Die Testanwendung wird hier bei der Frage nach Anwendungen vom Typ `ear` gefunden.

4. **Aus welchen Arten von Anwendungen besteht die analysierte Applikation?**

   Erwartetes Ergebnis:

   war

   Geliefertes Ergebnis:

   war

   Erläuterung:

   Wie in der vorhergegangenen Frage muss hier die Frage aufgespalten werden für jeden möglichen Typ (`jar`, `war`, `ear` und `rar`). Dann liefert die Frage das richtige Ergebnis.

5. **Welche Java EE APIs verwendet die Anwendung?**

   Erwartetes Ergebnis:

   TestAPIKlasse

   Geliefertes Ergebnis:

   TestAPIKlasse

   Erläuterung:

   Mit dieser Antwort wird die Frage korrekt beantwortet. Es ist davon auszugehen, dass die Klasse *TestAPIKlasse* eine Klasse der API „TestAPI“ ist. Damit ist gezeigt, dass die Anwendung die API „TestAPI“ verwendet. Die Abfrage hat alle *benötigt Klasse*-Relationen von *TestEARApp* und deren Teilapplikationen herausgesucht.

6. **Welche Eigenschaften einer Anwendung sind mit einem speziellen Server kompatibel?**

   Erwartetes Ergebnis:

   TestEARJEApp, TestWARJEWeb

   Geliefertes Ergebnis:

   $\emptyset$
   beziehungsweise
   TestEARJEApp, TestWARJEWeb

   Erläuterung:

   Wegen der Open-World Annahme von Ontologien kann diese Frage nicht korrekt beantwortet werden. Das liegt daran, dass niemals alle *Java EE Attribute* von *Java EE Eigenschaften* geprüft werden können, weil es immer noch mehr *Java EE Attribute* geben kann. Wenn man die Frage allerdings dahingehend abwandelt, nach *Java EE Eigenschaften* zu suchen, die mindestens ein *Java EE Attribut* besitzen, das mit dem Server kompatibel ist, dann liefert die Frage die richtigen Ergebnisse.

Die nachfolgenden Fragen können ebenso wie die vorherige Frage „Welche Eigenschaften einer Anwendung sind mit einem speziellen Server kompatibel?“ nicht richtig beantwortet werden und liefern keine Ergebnisse. Das liegt, wie schon zuvor, an der Open-World Annahme einer Ontologie (vgl. Abschnitt 6.1.6).

7. **Welche Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?**

   Hier muss eine ähnliche Anfrage gestellt werden wie bei der vorherigen Frage, allerdings mit einer Negation. Das liefert wegen dem Open-World Prinzip keine Ergebnisse.

8. **Wie viele Eigenschaften einer Anwendung sind mit einem speziellen Server nicht kompatibel?**

   Diese Frage ist eine Folgerung aus der vorherigen Frage, kann daher ebenfalls nicht zuverlässig beantwortet werden.

9. **Kann die Anwendung auf einem speziellen Server funktionieren?**

   Hier muss die in der vorherigen Frage errechnete Anzahl geprüft werden und zusätzlich werden noch die verwendeten *Java Klassen* überprüft. Der erste Teil kann nicht sicher beantwortet werden und im zweiten Teil wird auch auf Nichtexistenz geprüft. Dies ist ebenfalls nicht sicher, weshalb keine Ergebnisse geliefert werden.

10. **Auf welchen Anwendungsservern ist die analysierte Anwendung lauffähig?**

    Diese Frage ist eine Folgerung aus der vorherigen Frage mit offenem *Anwendungsserver*. Sie ist somit ebenfalls nicht sicher beantwortbar.

11. **Welcher Server wäre der geeignetste für die analysierte Anwendung?**

    Diese Frage ist eine weitere Verallgemeinerung der vorherigen Fragen und damit auch nicht sicher lösbar.

6.2.3 Zusammenfassung

Zusammengefasst lässt sich feststellen, dass sich einige der Kompetenzfragen mit Anfragen an die Ontologie beantworten lassen. Die wirklich interessanten Fragen bezüglich der Kompatibilität zwischen Applikation und Anwendungsserver können allerdings aufgrund der Open-World Annahme von Ontologien nicht sicher beantwortet werden.
Wenn man bei den Fragen die Open-World Annahme berücksichtigt, liefern alle das erwartete Ergebnis. Dieses Ergebnis ist in den meisten Fällen die leere Menge. Die Aussagekraft dieser Lösung ist somit nicht besonders hoch. Aus diesem Grund kann nicht direkt mit einer Ontologie und deren internen Mechanismen gearbeitet werden. Die Lösung hierfür ist die Entwicklung einer Software, die die Ontologie als Input erhält und auf dieser Struktur über die vorhandenen Regeln und Axiome Schlussfolgerungen zieht. Dies muss allerdings unter der Closed-World Annahme geschehen. Zusätzlich soll dieses Programm auch die Analysefunktionen implementieren und einen abschließenden Report über die Migrierbarkeit eine analysierten Applikation auf einen speziellen Server liefern. In Kapitel 7 wird eine solche Software implementiert und vorgestellt.

7 Prototyp zur Java EE Applikationsmigration

In diesem Kapitel wird unter Berücksichtigung der bisherigen Ergebnisse die Implementierung eines Prototypen vorgestellt. Zuerst wird in Kapitel 7.1 auf die einzelnen Programmteile der Implementierung eingegangen. Anschließend beschreibt Kapitel 7.2 die verwendeten Technologien. Kapitel 7.3 beinhaltet Details zur Implementierung und Probleme, die während der Entwicklung aufgetreten sind. In Kapitel 7.4 wird die Verwendung des Prototypen beschrieben und erläutert wie dieser konfiguriert und erweitert werden kann. Kapitel 7.5 beschäftigt sich mit der beispielhaften Analyse einer Anwendung durch den Prototypen und welche Resultate geliefert werden. In Kapitel 7.6 werden alle Ergebnisse der Implementierung zusammengefasst und Schlussfolgerungen daraus gezogen.

7.1 Bestandteile

In diesem Kapitel werden die Bestandteile des Prototypen beschrieben. Hierzu gehören neben dem Hauptprogramm auch Hilfsprogramme zur Analyse von Java EE Anwendungsservern und Deployment Deskriptor XML Schemata.
In Abbildung 7.1 sind alle Bestandteile des Prototypen dargestellt. Das Hauptprogramm analysiert das Applikationspaket, das in der Input Ontologie beschrieben ist und speichert die Informationen in der Basisontologie ab. Anschließend werden diese Informationen mit den Server Ontologien verglichen und die Resultate als HTML-Seite präsentiert. Das Hilfsprogramm „Deployment Deskriptor Analyse“ analysiert mit den unterstützten XSD- oder DTD-Schemata eines Anwendungsservers die Java EE Eigenschaften und liefert die Server Deployment Deskriptor Ontologie an den Prototypen. Zusätzlich läuft das Hilfsprogramm „Classpath Analyse“ auf einem Anwendungsserver und gibt die Ontologie mit Server Classpath Elementen als Ausgabe an den Prototypen weiter.

7.1.1 Hauptprogramm

Als Hauptprogramm wird der Teil des Prototyps bezeichnet, der Java EE Applikation analysiert und mit den Eigenschaften eines Java EE Anwendungsservers vergleicht. Als Ergebnis soll hierbei eine HTML-Seite erzeugt werden, das die gefundenen Probleme der Applikation mit jedem einzelnen Anwendungsserver auflistet.
Es soll ein komplettes Anwendungspaket in die zuvor entwickelte Ontologie geladen werden und mit den Eigenschaften der zu prüfenden Anwendungsserver verglichen werden. Für die Analyse des Anwendungspakets stehen zwei Standardanalysefunktionen zur Verfügung, die in Kapitel 7.3 genauer beschrieben werden. Hierbei handelt es sich zum Einen um eine Funktion zur strukturellen Analyse eines Anwendungspakets und zur Klassifizierung der gefundenen

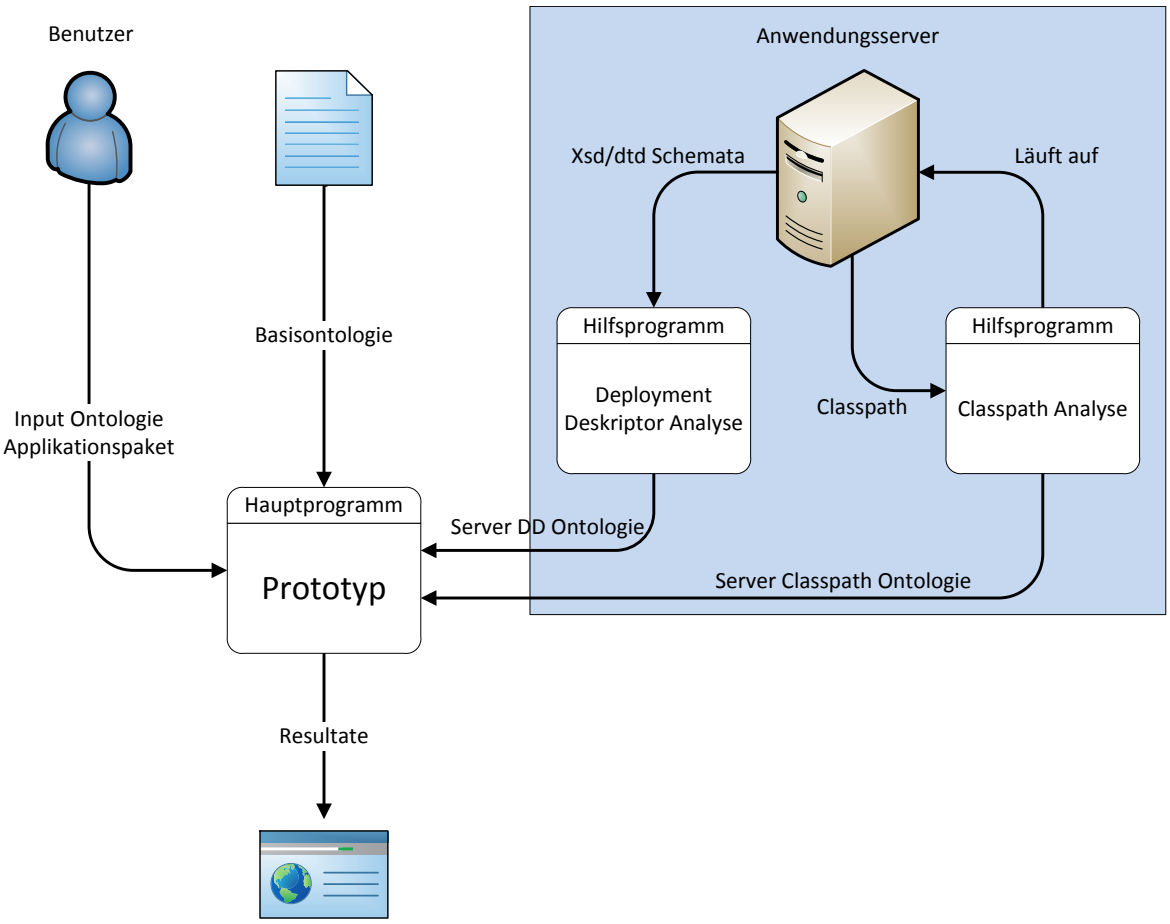


Abbildung 7.1: Veranschaulichung der Programmbestandteile des Prototypen bestehend aus dem Hauptprogramm und den beiden Hilfsprogrammen zur Analyse des Server Classpaths und der unterstützten Deployment Deskriptoren.

Dateien, zum Anderen um eine Funktion, die Deployment Deskriptoren analysiert und die gefundenen Parameter in der Ontologie abspeichert.
Anschließend werden die gefundenen Parameter mit denen des Anwendungsservers verglichen und Inkompatibilitäten aufgezeigt. Diese werden in anschaulicher Form als HTML-Seite präsentiert.

7.1.2 Analyse von Deployment Deskriptoren Schemata

Deployment Deskriptoren haben im Allgemeinen eine *XML Schema Definition* (XSD). Für ältere Java EE Versionen sind statt XSD auch *Dokumenttypdefinitionen* (DTD) angegeben [Orad]. Diese sind online verfügbar und können herangezogen werden, damit alle möglichen Parameter in einem Deployment Deskriptor bestimmt werden können.
Um die mögliche Konfiguration eines Anwendungsservers zu erhalten, müssen alle XSDs und DTDs, der von diesem Server unterstützten Deployment Deskriptoren, untersucht und in Ontologieform gespeichert werden. Zusätzlich zu den normalen Java EE Deployment Deskriptoren können manche Anwendungsserver auch individuelle Deskriptoren unterstützen.

Für einige Anwendungsserver wurden diese in Kapitel 2.3 beschrieben.
Daher ist zusätzlich zum Hauptprogramm ein Tool erforderlich, das anhand von XSDs und DTDs die möglichen Java EE Eigenschaften und Attribute zu einem Anwendungsserver hinzufügt. Die dabei gefundenen Informationen müssen anschließend in einer OWL Ontologie gespeichert werden, damit sie vom Hauptprogramm verwendet werden können.

7.1.3 Analyse des Classpaths von Java EE Anwendungsservern

Ein Anwendungsserver bringt die von ihm unterstützten Java EE APIs als Implementierung in Form von Klassen mit. Diese sind von der ausgeführten Anwendung erreichbar und können somit dort verwendet werden. Um alle möglichen Klassen eines Anwendungsservers herauszufinden, müssen alle von einer Anwendung verwendbaren Klassen aufgelistet werden. Aus diesem Grund ist es notwendig eine Webanwendung zu implementieren, die auf einem Anwendungsserver ausgeführt werden kann. Diese Webanwendung analysiert alle von ihr erreichbaren Klassen und speichert diese anschließend in einer OWL Ontologie ab, damit diese Informationen vom Hauptprogramm verwendet werden können.

7.2 verwendete Technologien

In diesem Kapitel werden die Technologien vorgestellt, die für die Implementierung des Prototypen verwendet wurden. Hierbei wird zwischen Technologien, die die Entwicklung des Prototyps betreffen, und denen, die die Ausführung des Prototyps betreffen, unterschieden.

7.2.1 Entwicklung

In diesem Kapitel werden die Technologien beschrieben, die zur Entwicklung des Prototypen herangezogen wurden.

Java

Für die Implementierung des Prototypen wird auf Java in den Versionen 7 und 8 zurückgegriffen. Die Verwendung von Java 7 ist nötig, um die Webanwendung zur Analyse des Classpaths auch auf GlassFish 3 ausführen zu können. Im Hauptprogramm und dem Analysetool für Deployment Deskriptoren Schemata wird Java 8 als neueste Java Version verwendet.

Eclipse SWT

Für die Oberfläche der Zusatztools wird Eclipse SWT verwendet. SWT steht für *Standard Widget Toolkit* und stellt eine vom Betriebssystem unabhängige Schnittstelle für grafische Benutzeroberflächen zur Verfügung. Das Tool zur Analyse von Deployment Deskriptor Schemata besitzt zur einfacheren Bedienung eine Oberfläche mittels SWT. [RB]

7.2.2 Ausführung

Dieses Kapitel beschäftigt sich mit den Technologien, die zur Ausführung des Prototypen verwendet werden.

OWL API

Die OWL API ist eine Java Schnittstelle, um OWL Ontologien zu erstellen, manipulieren und serialisieren. Gleichzeitig ist dies auch die Referenzimplementierung. Auf OWL wird in jedem Programmteil zurückgegriffen, um Informationen entweder in einer Ontologie abzulegen, diese zu laden oder zu speichern. Über die sogenannten Reasoner, hier in der Ausprägung *Pellet*, können Daten in der Ontologie abgefragt werden, wobei zuerst alle Axiome und Regeln geladen und angewendet werden müssen. [Sou]

Java Decompiler – jd

Für die Analyse von kompiliertem Java Code ist es nötig, diesen zu dekompilieren, damit die verwendeten Klassen herausgefunden werden können. Der Java Decompiler *jd* stellt eine einfache Möglichkeit zur Verfügung, um einzelne Klassen oder komplette Java Pakete zu dekompilieren. [Dup]

Reflections

Reflections ist ein Tool, um den Classpath während der Laufzeit eines Programms abzufragen. Es wird in der Webanwendung zur Analyse des Classpaths von Anwendungsservern verwendet, um Subtypen der Klasse `Object` zu finden. [ron]

Apache Xerces2

Apache Xerces2 ist der XML Parser, der in den Java Distributionen ab Version 7u40 integriert ist. Zusätzlich ist es möglich, XSD-Dokumente einzulesen. Genau dieser Mechanismus wird im Zusammenhang mit dem Einlesen von Deployment Deskriptor Schemata verwendet. [Apae]

TRANG

Trang ist eine Java Bibliothek, die zwischen verschiedenen Arten von XML-Schema-Typen konvertieren kann. Da es keine einfache Möglichkeit gibt, DTD-Dateien zu analysieren, wird jede solche Datei in ein XSD-Dokument umgewandelt und dann weiter analysiert. [Ltd]

7.3 Implementierungsdetails

In diesem Kapitel werden die wichtigsten beiden Aspekte des Prototypen (`Function` und `MigrationContext`) beschrieben und auf die Implementierung näher eingegangen. Zusätzlich wird in Abschnitt 7.3.3 auf aufgetretene Probleme eingegangen und wie diese umgangen werden konnten.

7.3.1 Function

Eine `Function` ist ein `Interface` für alle zur Migration relevanten Funktionen, die nacheinander abgearbeitet werden sollen.
In Abbildung 7.2 ist das Interface `Function` mit den beiden Implementierungen `StructureFunction` und `DDElementFunction` zu sehen. Diese sind die elementaren Analysefunktionen, die implementiert wurden. Nachfolgend werden die verwendeten Methoden kurz beschrieben:

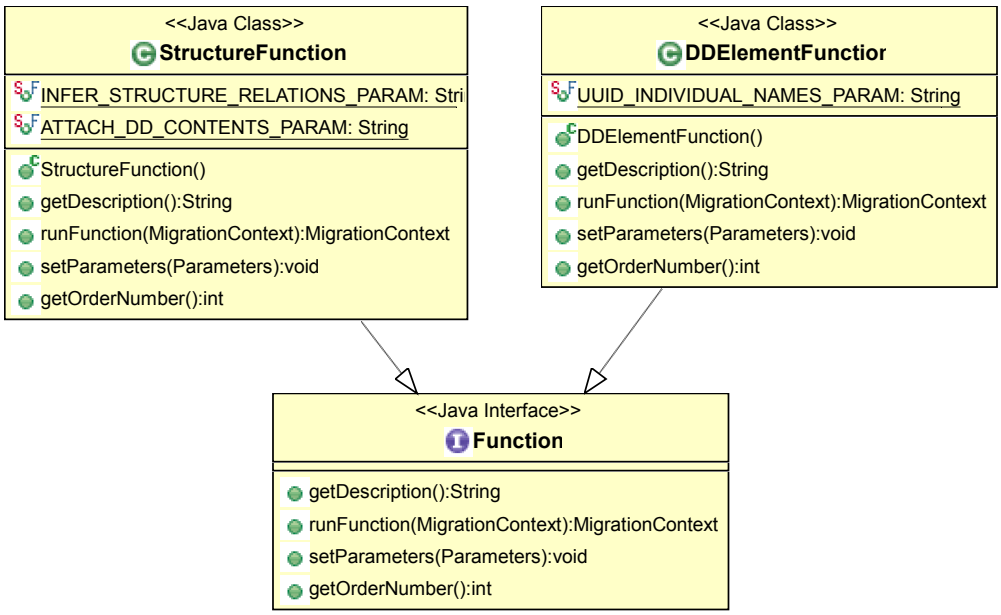


Abbildung 7.2: Klassendiagramm des Interfaces `Function` mit den beiden Implementierungen `StructureFunction` und `DDElementFunction`.

`getDescription():String`

Mit der Methode `getDescription` soll eine Beschreibung der Funktion zurückgegeben werden, damit im Log genauer zu sehen ist, welche Art von Funktion zu welchem Zeitpunkt durchgelaufen ist. Dies dient hauptsächlich der Übersichtlichkeit.

`getOrderNumber():int`

Mit der Methode `getOrderNumber` wird eine Zahl zurückgegeben, die die Funktionen in eine Reihenfolge bringt. Dies ist wichtig, wenn unterschiedliche Funktionen auf den Ergebnissen einer anderen Funktion aufbauen. So ist sichergestellt, dass eine Funktion mit einer niedrigeren Zahl vor einer anderen mit höherer Zahl durchläuft.

`runFunction(MigrationContext):MigrationContext`

`runFunction` ist die Hauptmethode einer Funktion. Hiermit wird der gewünschte Umfang der Funktion ausgeführt und Daten in der Ontologie gespeichert oder abgefragt. Hierbei ist die Klasse `MigrationContext` der Übergabe- und Rückgabewert (siehe Abschnitt 7.3.2).

`setParameters(Parameters):void`

Die Methode `setParameters` kann für jede Funktion aufgerufen werden, um Parameter zu setzen, die andere Ergebnisse liefern können. Ein Beispiel hierfür ist der Parameter `INFER_STRUCTURE_RELATIONS_PARAM` der Function `StructureFunction`, mit dem festgelegt wird, ob alle schlussfolgernden Strukturrelationen wie *Datei enthält* direkt angelegt werden sollen.

Für die Analyse von Applikationspaketen stehen die beiden in Abbildung 7.2 gezeigten Funktionen `StructureFunction` und `DDElementFunction` zur Verfügung. Die Funktion `Struc-`

tureFunction muss vor der DDElementFunction gestartet werden, da hier die Daten der Deployment Deskriptoren in den MigrationContext geladen werden.
StructureFunction analysiert die Struktur eines Applikationspakets, klassifiziert die Dateien (*Datei*, *Java Paket*, *Anwendungspaket*, *Verzeichnis*, *Java Klasse* und *Deployment Deskriptor*) und speichert diese Informationen im MigrationContext ab. Zusätzlich kann über den Parameter ATTACH_DD_CONTENTS_PARAM gesteuert werden, ob die Dateiinhalte von Deployment Deskriptoren an den MigrationContext angehängt werden sollen. Mit dem Parameter INFER_STRUCTURE_RELATIONS_PARAM kann festgelegt werden, ob alle Strukturrelationen manuell angelegt werden sollen oder nicht.
Die Funktion DDElementFunction analysiert gefundene Deployment Deskriptoren und speichert die Java EE Eigenschaften und Attribute im MigrationContext ab. Über den Parameter UUID_INDIVIDUAL_NAMES_PARAM kann gesteuert werden, ob jede neu gefundene Eigenschaft und jedes Attribute eine eindeutige UUID bekommt. Auf diese Art können alle Eigenschaften und Attribute unterschieden werden. Der Nachteil ist, dass die Analyse des gleichen Deployment Deskriptors bei mehreren Durchläufen unterschiedliche Ergebnisse liefert. So sind die Ergebnisse nicht mehr trivial vergleichbar.

7.3.2 MigrationContext

Der MigrationContext ist das Objekt, das die Schnittstelle zwischen den einzelnen Funktionen und der Ontologie als Datenspeicher darstellt.
In Abbildung 7.3 ist das Klassendiagramm der Klasse MigrationContext zu sehen. MigrationContext erbt von Context. Dieses Design wird deshalb gewählt, damit die Zusatzprogramme keinen zusätzlichen Umfang bekamen, aber den Context dennoch verwenden können. In der Klasse Context sind Inkompatibilitäten bezüglich einzelner Server ebenso wie ein Basisverzeichnis oder der Dateiinhalt von einzelnen Dateien enthalten. Diese Elemente werden zwar im Hauptprogramm benötigt, in den Hilfsprogrammen allerdings nicht. Im folgenden werden die Methoden von Context und MigrationContext kurz angesprochen:

`Context()`

Der Konstruktor der Klasse Context benötigt keine Übergabeparameter. Während der Initialisierung von Context wird die grundlegende Ontologie geladen.

`MigrationContext(String, Collection<String>)`

Der Konstruktor der Klasse MigrationContext benötigt zwei Übergabeparameter. Der erste Übergabeparameter ist das Grundverzeichnis der Migration, in dem die Eingabeontologie und Serverontologien hinterlegt sind und auch die Ausgabe stattfindet. Der zweite Übergabeparameter ist die Liste der Servernamen, für die die Analysen durchgeführt werden sollen. Zusätzlich wird wie bei Context die grundlegende Ontologie geladen.

`addAxiomsToOntology(Set<OWLAxiom>):void`

Über die Methode addAxiomsToOntology können Änderungen an der Ontologie vorgenommen werden. Jede Zuordnung wie beispielsweise zu einer Klasse oder eines Attributwerts in der Ontologie wird mit einem OWLAxiom definiert. Eine OWLOntology ist nichts anderes als eine Sammlung von vielen Instanzen der Klasse OWLAxiom. OWLAxiom ist ein Bestandteil der OWL API.

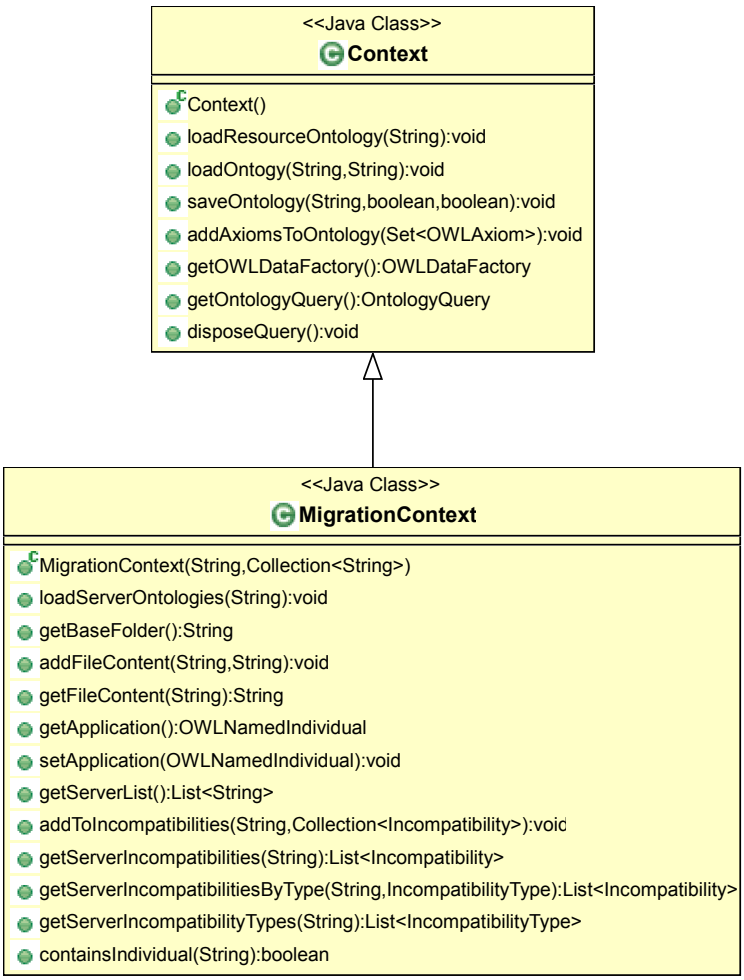


Abbildung 7.3: Klassendiagramm der Klasse `MigrationContext`, die von der Klasse `Context` erbt.

`addFileContent(String, String):void`

Mit der Methode `addFileContent` wird der Inhalt einer Datei an den `MigrationContext` angehängt. Die Datei wird hierbei anhand der ID in der Ontologie identifiziert. Der erste Parameter ist die ID der Datei und der zweite der Dateiinhalt. Diese Funktionalität wird benötigt, da Dateien nur während einer Funktion vorhanden sind und auf diese Weise auch von anderen Funktionen auf Dateiinhalte zugegriffen werden kann.

`addToIncompatibilities(String, Collection<Incompatibility>):void`

In der Klasse `MigrationContext` werden die Inkompatibilitäten der analysierten Applikation mit unterschiedlichen Servern festgehalten. Mit der Methode `addToIncompatibilities` kann einem Server (repräsentiert durch einen String-Wert) eine Sammlung von Inkompatibilitäten zugewiesen werden.

`containsIndividual(String):boolean`

Mit der Methode `containsIndividual` kann die Existenz einer beliebigen Entität in der Ontologie geprüft werden. Nötig ist hier die ID der Entität in Form eines String-Werts.

`disposeQuery():void`

Das Hinzufügen und Laden von neuen Elementen in die Ontologie, führt dazu, dass der Reasoner nicht mehr den aktuellen Stand der Ontologie besitzt. In diesem Fall kann es nötig sein, den Reasoner neu zu laden. Hierfür kann über die Methode `disposeQuery` der Reasoner zurückgesetzt werden.

`getApplication():OWLNamedIndividual`

Mit der Methode `getApplication` wird die Entität der Ontologie (`OWLNamedIndividual`) zurückgegeben, das der analysierten Applikation entspricht. Eine der ersten Schritte im Migrationsprozess ist das Setzen der Entität. Die Klasse `OWLNamedIndividual` ist ein Bestandteil der OWL API und spiegelt eine Entität einer Ontologie wieder.

`getBaseFolder():String`

Durch Aufrufen der Methode `getBaseFolder` wird das zu Beginn der Migration gesetzte Basisverzeichnis zurückgegeben.

`getFileContent(String):String`

Die Methode `getFileContent` liefert den Inhalt einer eingelesenen Datei zurück, wenn diese über `addFileContent` vorher hinzugefügt wurde. Der Übergabeparameter entspricht der ID der gewünschten Datei.

`getOWLDataFactory():OWLDataFactory`

Mit der Methode `getOWLDataFactory` wird die zur hinterlegten Ontologie gehörende `OWLDataFactory` zurückgegeben. Die `OWLDataFactory` ist ein Teil der OWL API und dient der Erstellung von Entitäten und Axiomen (`OWLAxiom`).

`getOntologyQuery():OntologyQuery`

Die Methode `getOntologyQuery` liefert eine Instanz der Schnittstelle zum Reasoner zurück, mit der die Ontologie abgefragt werden kann. Falls aktuell kein Reasoner geladen ist, wird ein neuer Reasoner geladen.

`getServerIncompatibilities(String):List<Incompatibility>`

Die Methode `getServerIncompatibilities` liefert alle Inkompatibilitäten für einen Server zurück. Der einzige Eingabeparameter ist der Servername.

`getServerIncompatibilitiesByType(String, IncompatibilityType):List<Incompatibility>`

Mit der Methode `getServerIncompatibilitiesByType` werden alle Inkompatibilitäten eines speziellen Servers und Typs zurückgeliefert.

`getServerIncompatibilityTypes(String):List<IncompatibilityType>`

Inkompatibilitäten haben einen festgelegten Typ. Diese sind *Classpath*, *Property* oder *Attribute* für Inkompatibilität bezüglich des Classpaths, Eigenschaften oder Attribute. Die Methode `getServerIncompatibilityTypes` liefert alle registrierten Typen von Inkompatibilitäten für einen speziellen Server zurück.

`getServerList():List<String>`

Mit der Methode `getServerList` wird die Liste von Servernamen zurückgegeben, mit denen der `MigrationContext` initialisiert wurde.

`loadOntogy(String, String):void`

Mit der Methode `loadOntology` kann eine im OWL Format abgespeicherte Ontologie in die Klasse `Context` geladen werden. Hierbei ist der erste Parameter der Verzeichnispfad zur Ontologie und der zweite Parameter der Dateiname der Ontologie.

`loadResourceOntology(String):void`

Die Methode `loadResourceOntology` ist ähnlich zu `loadOntology`, allerdings muss sich hier die mit dem Eingabeparameter identifizierte Ontologie auf dem Classpath der Anwendung befinden.

`loadServerOntologies(String):void`

Mit der Methode `loadServerOntologies` können alle im OWL Format gespeicherten Ontologien in einem Ordner geladen werden. Der Ordnername ist der einzige Eingabeparameter und dieser Ordner muss sich im Basisverzeichnis befinden.

`saveOntology(String, boolean, boolean):void`

Mit der Methode `saveOntology` kann die im `Context` enthaltene Ontologie gespeichert werden. Hierbei ist der erste Parameter der Ausgabepfad. Der zweite Parameter gibt an, ob vor dem Speichern noch Schlussfolgerungen gezogen werden sollen. Das bedeutet bei großen Datenmengen einen hohen Zeitfaktor, da jedes in der Ontologie enthaltene Axiom neu geschlussfolgert werden muss, darunter auch beispielsweise Subklassen-Axiome. Der dritte Parameter gibt an, ob die Ontologie im Format RDF oder im Format OWL abgespeichert werden soll.

`setApplication(OWLNamedIndividual):void`

Mit der Methode `setApplication` kann einmalig ein `OWLNamedIndividual` als analysierte Applikation gesetzt werden.

7.3.3 Aufgetretene Probleme

In diesem Kapitel werden Probleme, die während der Implementierung des Prototypen aufgetreten sind, und ihre Lösungen angesprochen.

7.3.3.1 Anwendung von Regeln

Das erste aufgetretene Problem betrifft die in Abschnitt 5.2.10 definierten Regeln. Bei großen Ontologien mit sehr vielen Instanzen und Relationen dauert das Anwenden von Regeln signifikant länger. Das liegt an der Implementierung der Reasoner, die bei jeder Änderung der Ontologie nochmals alle Regeln überprüfen.
Die Konsequenz dieses Problems ist, dass im Hauptprogramm keine einzige Regel verwendet wird. Die benötigten Informationen wie das Setzen der Relation *Datei enthalten in* wird manuell in `StructureFunction` implementiert, was außer zusätzlichem Speicherbedarf beim

Zwischenspeichern der benötigten Instanzen keine zusätzlichen Nachteile besitzt und signifikant schneller arbeitet.
In Kapitel 7.6 wird näher auf dieses Problem und seine Auswirkungen auf die Verwendung einer Ontologie in der Migration von Java EE Applikationen eingegangen.

7.3.3.2 Reasonerperformance

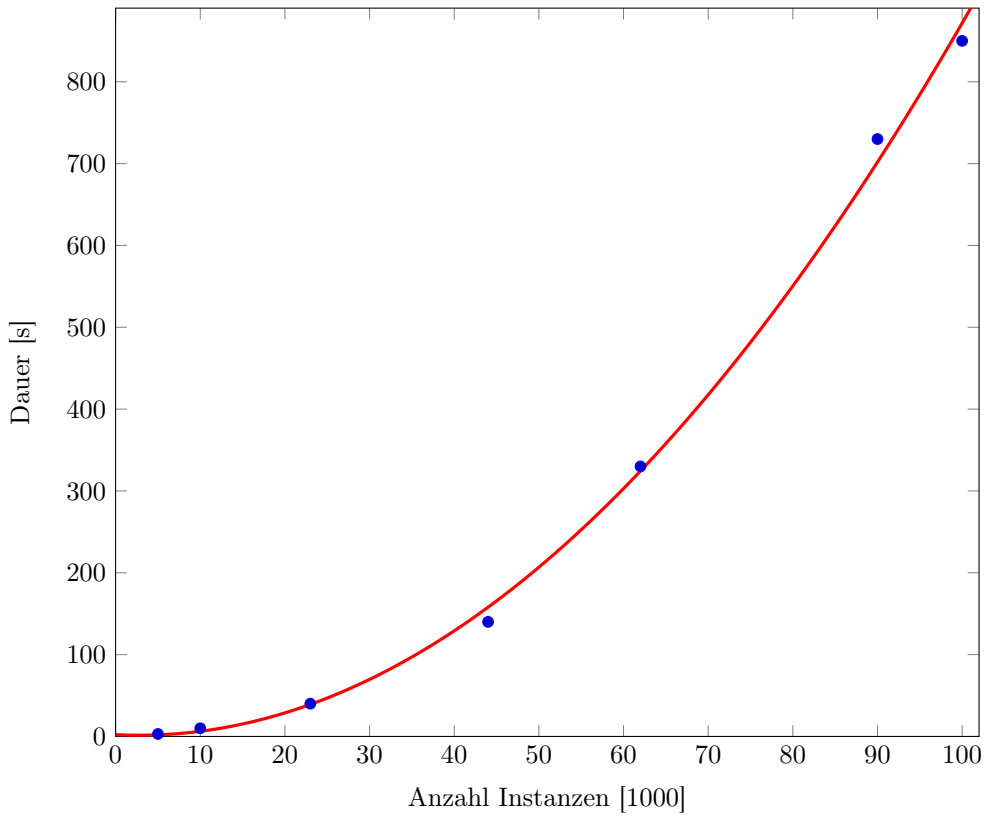


Abbildung 7.4: Neuladedauer in Sekunden des Reasoners beim Laden einer Ontologie. Die Punkte stellen Messungen bei 5.000, 10.000, 23.000, 44.000, 63.000, 90.000 und 100.000 Entitäten dar. Die Linie ist die quadratische Regressionsparabel für diese Messpunkte.

Die Classpath-Analyse des Oracle WebLogic Servers in Version 12.1.3 erbringt insgesamt über 90.000 Java Klassen. Zusammen mit den Java EE Eigenschaften der analysierten Deployment Deskriptor Schemata erreicht man etwa 100.000 Instanzen. Das Einlesen einer solchen gewaltigen Ontologie erfolgt in wenigen Sekunden. Problematisch ist hingegen das benötigte Neustarten des Reasoners, damit die Instanzen des WebLogic Servers auch gefunden werden können. Diese 100.000 Instanzen werden in deutlich über 10 Minuten in den Reasoner geladen. Im Vergleich dazu benötigt das Neustarten des Reasoners für einen Apache Tomcat in Version 7.0.65 mit rund 10.000 Instanzen nur 10 Sekunden. Das Neustarten des Reasoners bei ungefähr 5.000 Instanzen benötigt circa 2 Sekunden.
Abbildung 7.4 zeigt mehrere Neuladezeiten des Reasoners mit unterschiedlicher Anzahl an Entitäten. Es ist deutlich der quadratische Verlauf der Neuladedauer zu sehen. Die Linie

stellt die anhand der Messpunkte erstellte Regressionsparabel dar.
Wegen der hohen Neuladezeit bei vielen Entitäten in der Ontologie ist es nicht empfehlenswert einen Programmdurchlauf mit mehr als einem Anwendungsserver gleichzeitig zu starten. Das zwangsläufig durchzuführende Neuladen des Reasoners benötigt mehr Zeit als die Server jeweils einzeln in unterschiedlichen Durchläufen zu vergleichen.

7.4 Inbetriebnahme

Dieses Kapitel beschäftigt sich mit dem Betrieb des Prototypen, welche Eingabeparameter benötigt werden und wie der Prototyp erweitert werden kann.

7.4.1 Eingabeparameter

Der Prototyp besitzt zwei essentielle Eingabeparameter und einige optionale, die der folgenden Aufzählung entnommen werden können:

`-baseDirectory <Verzeichnispfad>`

Mit dem Parameter `-baseDirectory` wird das Basisverzeichnis übergeben, in dem sich die Eingabeontologie, die Serverontologien und das zu untersuchende Anwendungspaket befinden müssen. In diesem Verzeichnis werden zusätzlich die Ergebnisse der Migration gespeichert.

`-inputOwl <Dateiname>`

Der Parameter `-inputOwl` gibt an, welche Datei als Eingabeontologie geladen werden soll. In dieser Ontologie werden die Analysefunktionen mit ihren Klassen definiert und das Anwendungspaket mit dem Dateiname beschrieben. Ebenso wie das Anwendungspaket, muss sich die Eingabeontologie im Basisverzeichnis befinden. Das zu analysierende Anwendungspaket muss die ID „MainAppPackage“ besitzen. Nur dann kann eine Analyse richtig durchgeführt werden.

`-server <Servername>`

Der Parameter `-server` kann mehrfach verwendet werden und fügt dem `MigrationContext` jeweils einen weiteren zu vergleichenden Anwendungsserver hinzu. Für jeden dieser Server müssen sich im Basisverzeichnis einzelne Ordner befinden, in denen die Eigenschaften des Anwendungsservers hinterlegt sind.

`-standardCompair <true|false>`

Über den Parameter `-standardCompair` wird gesteuert, ob ein normaler Vergleich für das analysierte Anwendungspaket durchgeführt werden soll. Ein normaler Vergleich besteht aus der Analyse der Java EE Eigenschaften und Attribute, sowie dem Vergleich der Classpath Elemente zwischen Anwendungspaket und Anwendungsserver.

Der Prototyp liegt als Java Paket (`jar`-Datei) vor und kann direkt über das Javakommando `java -jar` mit den entsprechenden Parametern ausgeführt werden.

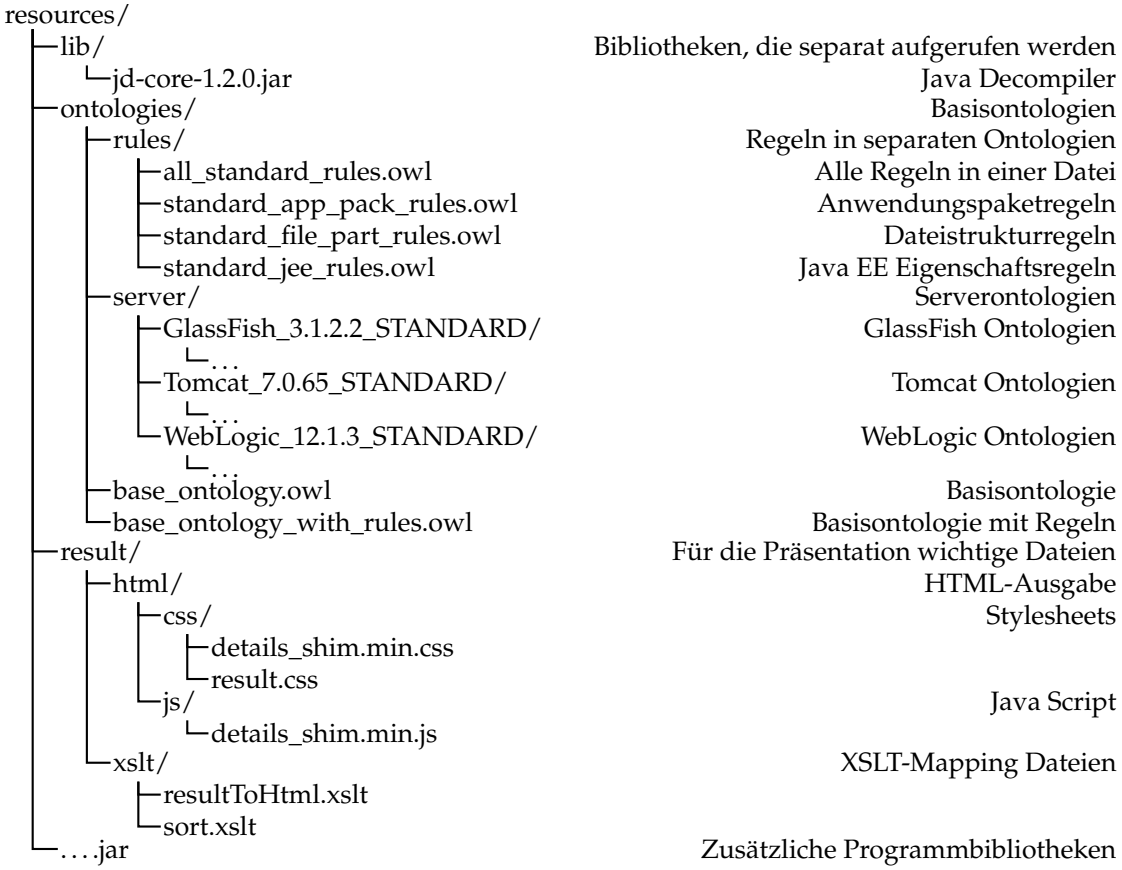


Abbildung 7.5: Die Struktur des Ordners „resources“, in dem die für den Prototypen wichtigen Elemente hinterlegt sind.

7.4.2 Erweiterungen

Die relevanten Bibliotheken und zusätzlichen Ressourcen des Prototypen, wie beispielsweise die Grundontologie, liegen parallel zur `jar`-Datei im Ordner „resources“. Dieser Ordner hat die in Abbildung 7.5 gezeigte Struktur mit den Ordnern „lib“, „ontologies“ und „result“ für separat aufgerufene Bibliotheken, Basisontologien und Präsentation des Ergebnisses.
Um neue Funktionen hinzuzufügen reicht es die entsprechenden Funktionen zu implementieren und direkt im Ordner „resources“ abzulegen. Gleiches gilt für Änderungen an der Basisontologie, falls diese nötig wären.

7.5 Anwendungsbeispiel

Der Prototyp wurde anhand einiger Beispielanwendungen getestet. Hier wird ein Test mit der Java EE 6 Beispielanwendung „MailConnector Resource Adapter Sample Application“ vorgestellt. Dieses Beispiel besteht aus einer Enterprise Anwendung, die aus einem EJB und einer Webanwendung zusammengesetzt ist, und einem externen Ressourcen Adapter, auf den von der Enterprise Anwendung zugegriffen wird. Es wurde sich für dieses Beispiel entschieden, da es mehrere Komponenten des Java EE Standards in sich vereint und einige unterschiedliche Deployment Deskriptoren enthält. [Orac]

In diesem Kapitel werden zunächst die ausgewählten Applikationen beschrieben und ihre Charakteristika gesammelt. Anschließend wird eine Analyse für diese Applikationen mit dem Prototypen und ausgewählten Anwendungsservern durchgeführt. Abschließend werden die ermittelten Ergebnisse mit den erwarteten Ergebnissen verglichen.

7.5.1 Beispielapplikation

Die Beispielanwendung „MailConnector Resource Adapter Sample Application“ ist eine Applikation, die von Oracle bereitgestellt wird, um neue Elemente von Java EE 6 testen zu können. Da die Referenzimplementierung von Java EE 6 die Version 3 des Oracle GlassFish Servers ist, ist die Beispielanwendung explizit für GlassFish entwickelt worden und enthält GlassFish spezifische Deployment Deskriptoren.
Die Anwendung besteht aus den beiden Teilen `mailconnector.rar`, dem Ressourcen Adapter, und `mailconnector-ear.ear`, der Enterprise Anwendung:

`mailconnector.rar`

> `mailconnector.rar` kommt komplett ohne Deployment Deskriptor aus und besteht im Wesentlichen aus 22 verschiedenen Klassen, die die *Java EE Connector Architecture* (JCA) und *JavaMail API* verwenden. Die entsprechenden Klassen befinden sich hauptsächlich in den Packages `javax.resource` und `javax.mail`. Interessant ist die zusätzliche Verwendung der GlassFish proprietären Klasse `org.glassfish.security.common.PrincipalImpl` für die Absicherung des Mail-Connectors. Bezüglich dieser Absicherung werden noch zwei weitere Klassen aus dem Package `javax.security` verwendet.

`mailconnector-ear.ear`

> Die Enterprise Anwendung `mailconnector-ear.ear` besteht aus den beiden Teilanwendungen `mailconnector-ejb.jar` und `mailconnector-war.war` und wird über den Deployment Deskriptor `application.xml` konfiguriert. Ansonsten besteht es aus keinen eigenen Klassen.
>
> `mailconnector-ejb.jar`
>
> > `mailconnector-ejb.jar` ist als EJB in der Enterprise Anwendung `mailconnector-ear.ear` enthalten. Es wird durch die beiden Deployment Deskriptoren `ejb-jar.xml` und `sun-ejb-jar.xml` spezifiziert. Neben der obligatorischen Verwendung von EJB Klassen, wird auch JavaMail und der Ressourcen Adapter `mailconnector.rar` benutzt.
>
> `mailconnector-war.war`
>
> > Die Webanwendung `mailconnector-war.war` gehört zur Enterprise Anwendung `mailconnector-ear.ear` und wird mit den beiden Deployment Deskriptoren `web.xml` und `sun-web.xml` konfiguriert. Es werden hauptsächlich Klassen aus Java Servlet und JavaMail verwendet. Zusätzlich wird auf den Ressourcen Adapter `mailconnector.rar` zugegriffen.

Insgesamt werden die Beispielanwendungen durch die in Tabelle 7.1 aufgeführten Deployment Deskriptoren spezifiziert.
Die Anwendung ist auf GlassFish lauffähig, da sie hierfür erstellt wurde. Auf Apache Tomcat kann weder der Ressourcen Adapter noch die Enterprise Anwendung installiert werden.

Tabelle 7.1: Alle Deployment Deskriptoren, die in der Enterprise Anwendung `mailconnector-ear.ear` vorhanden sind.

Deployment Deskriptor	**Pfad innerhalb** `mailconnector-ear.ear`	**Beschreibung**
`application.xml`	`META-INF/application.xml`	Java EE Standard Deployment Deskriptor einer Enterprise Anwendung
`ejb-jar.xml`	`mailconnector-ejb.jar/META-INF/ejb-jar.xml`	Java EE Standard Deployment Deskriptor eines EJB
`sun-ejb.jar.xml`	`mailconnector-ejb.jar/META-INF/sun-ejb-jar.xml`	GlassFish spezifischer Deployment Deskriptor eines EJB
`web.xml`	`mailconnector-war.war/WEB-INF/web.xml`	Java EE Standard Deployment Deskriptor einer Webanwendung
`sun-web.xml`	`mailconnector-war.war/WEB-INF/sun-web.xml`	GlassFish spezifischer Deployment Deskriptor einer Webanwendung

Das liegt daran, dass Tomcat nur das Web Profile von Java EE implementiert. Entsprechend sind weder die JavaMail noch JCA oder EJB Klassen dort verfügbar. Zusätzlich wird ausschließlich der Deployment Deskriptor `web.xml` unterstützt. Die Anwendungsserver RedHat JBoss (WildFly), Oracle WebLogic und IBM WebSphere implementieren alle das Full Profile von Java EE, sind also prinzipiell fähig Ressourcen Adapter und Enterprise Anwendungen laufen zu lassen. Allerdings wird von keinem dieser Anwendungsserver der Deployment Deskriptor `sun-ejb-jar.xml` unterstützt. Nur WebLogic kann `sun-web.xml` interpretieren. Die GlassFish proprietäre Klasse `org.glassfish.security.common.PrincipalImpl` ist auf keinem dieser Server vorhanden. Deshalb kann der Ressourcen Adapter ebenso wie die Enterprise Anwendung auf keinem dieser Server ausgeführt werden.

7.5.2 Durchführung

Bei den Beispielanwendungen handelt es sich um Java EE 6 Anwendungen. Aus diesem Grund werden von jedem Servertyp Versionen für die Prüfung herangezogen, die diesen Standard unterstützen. Tabelle 7.2 zeigt die analysierten Anwendungsserver.

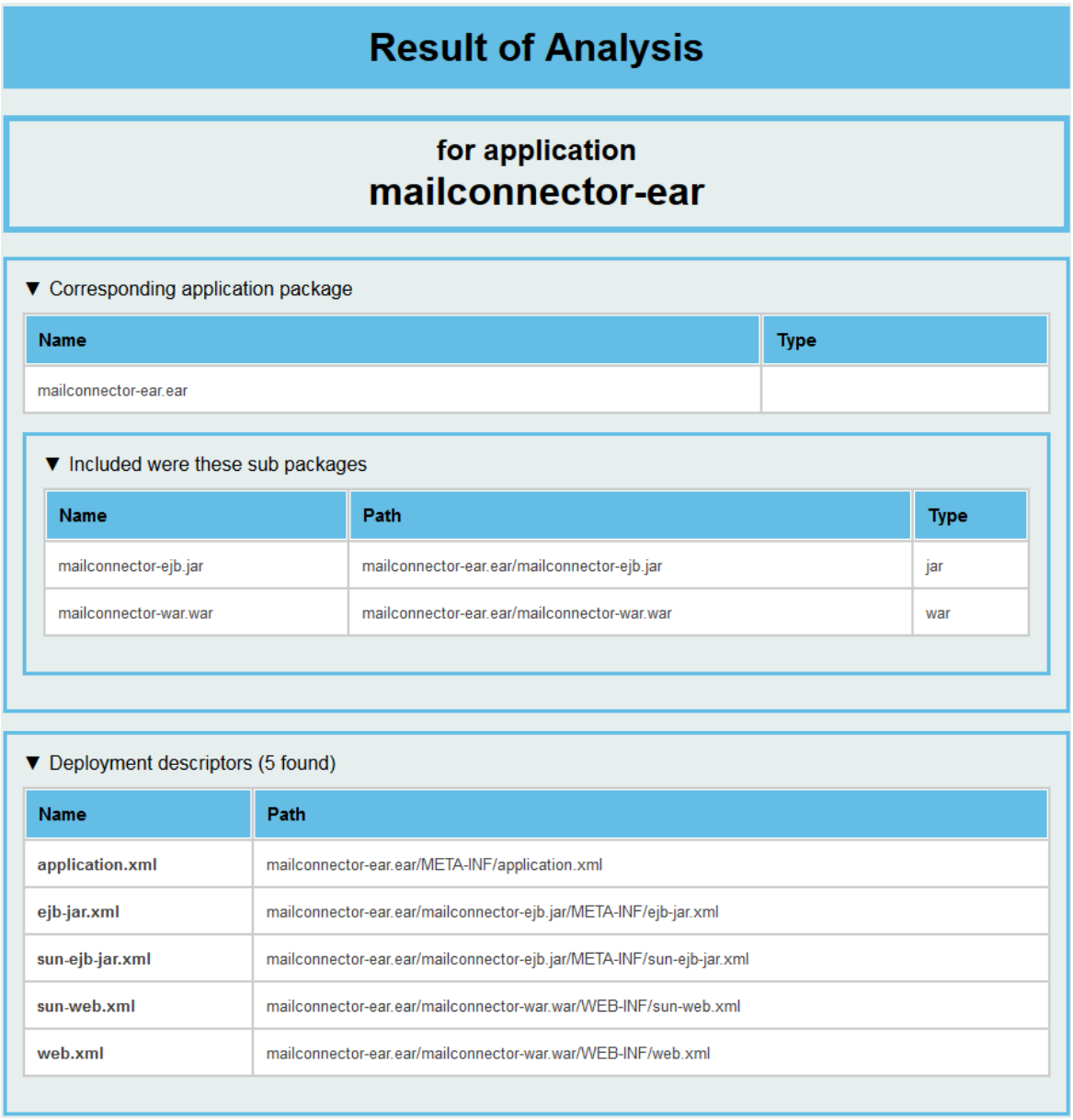
Tabelle 7.2: Die für die Prüfung analysierten Anwendungsserver mit den entsprechenden Versionen.

Anwendungsserver	**Version**	**Ausführung**
Apache Tomcat	7.0.65	STANDARD
IBM WebSphere	8.5.5.7	LIBERTY_FULL
RedHat WildFly	8.2.1	FULL_PROFILE
Oracle GlassFish	3.1.2.2	STANDARD
Oracle WebLogic	12.1.3	STANDARD

Die beiden Applikationen `mailconnector.rar` und `mailconnector-ear.ear` wurden in zwei separaten Programmläufen analysiert und jeweils mit den Servern verglichen.
Der Programmaufruf erfolgte jeweils mit den beiden Analysefunktionen `StuctureFunction` und `DDElementFunction`. Zusätzlich wurde ein Standardvergleich über den Classpath und die Java EE Eigenschaften durchgeführt. Für den Vergleich wurden die in Tabelle 7.2 gezeigten Server herangezogen. Der Programmaufruf geschieht folgendermaßen:

```
java -jar ServerMigration.jar
    -standardCompair true
    -inputOwl input.owl
    -baseDirectory "%cd%"
    -server Tomcat_7.0.65_STANDARD
    -server WebSphere_8.5.5.7_LIBERTY_FULL
    -server WildFly_8.2.1_FULL_PROFILE
    -server GlassFish_3.1.2.2_STANDARD
    -server WebLogic_12.1.3_STANDARD
```

Anschließend wurden die in einer HTML Seite aufgezeigten Ergebnisse mit den erwarteten Resultaten abgeglichen.
In Abbildung 7.6 sind die Ergebnisse des analysierten Anwendungspakets `mailconnector-ear.ear` dargestellt. Man sieht, dass die beiden Teilanwendungen `mailconnector-ejb.jar` und `mailconnector-war.war` erkannt wurden. Ebenso wurden die fünf Deployment Deskriptoren `application.xml`, `ejb-jar.xml`, `sun-ejb-jar.xml`, `sun-web.xml` und `web.xml` richtig entdeckt und mit dem entsprechenden Pfad angegeben.
Direkt unterhalb der Beschreibung der Anwendung befinden sich die Ergebnisse des Vergleichs der einzelnen Anwendungsserver. Dies wird hier beispielhaft für den Anwendungsserver IBM WebSphere 8.5.5.7 LIBERTY_FULL vorgestellt (Abbildung 7.7 und Abbildung 7.8). Es wurden für diesen Server vier CLasspath Probleme und zwei Java EE Eigenschaftsproblem gefunden.
Abbildung 7.7 zeigt, dass die Klasse `samples.connectors.mailconnector.servlet.MailBrowserServlet` aus der Webanwendung `mailconnector-war.war` die Klasse `samples.connectors.mailconnector.share.ConnectionSpecImpl` und die Klassen `JavaMailConnection` und `JavaMailConnectionFactory` aus dem Package `samples.connectors.mailconnector.api` verwendet. Zusätzlich benötigt die Klasse `samples.connectors.mailconnector.ejb.mdb.JavaMailMessageBean` aus dem EJB `mailconnector-ejb.jar` die Klasse `samples.connectors.mailconnector.api.JavaMailMessageListener`. Keine dieser benötigten Klassen ist auf dem Server Classpath oder in der Anwendung selbst zu finden. Das liegt daran, dass diese Klassen zum Ressourcen Adapter `mailconnector.war` gehören, also nicht in `mailconnector-ear.ear` zu finden sind. Der Ressourcen Adapter ist allerdings auf dem Anwendungsserver bekannt, so dass diese drei Probleme als irrelevant anzusehen sind.
Wie Abbildung 7.8 zeigt, sind die beiden Deployment Deskriptoren `sub-ejb-jar.xml` und `sun-web.xml` dem WebSphere Anwendungsserver nicht bekannt. Es wird jeweils der Hinweis gegeben, dass die Wurzelelemente nicht für diesen Server gefunden wurden, was darauf hindeutet, dass der komplette Deployment Deskriptor nicht unterstützt wird.
Insgesamt wurden die oben genannten vier Klassen bei allen Servern als Problem identifiziert. Zusätzlich wurden noch weitere zehn Classpath Probleme für Tomcat entdeckt, da dieser weder JavaMail noch EJBs unterstützt. Bezüglich der Deployment Deskriptoren wurden bei



Result of Analysis

for application
mailconnector-ear

▼ Corresponding application package

Name	Type
mailconnector-ear.ear	

▼ Included were these sub packages

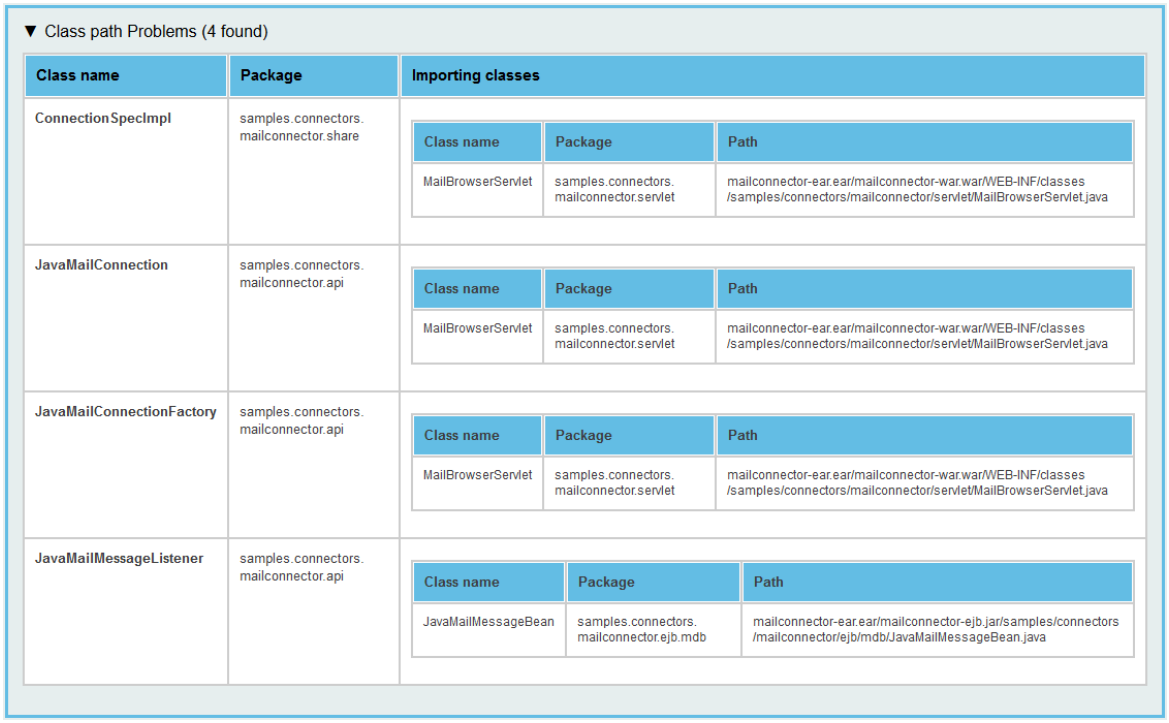
Name	Path	Type
mailconnector-ejb.jar	mailconnector-ear.ear/mailconnector-ejb.jar	jar
mailconnector-war.war	mailconnector-ear.ear/mailconnector-war.war	war

▼ Deployment descriptors (5 found)

Name	Path
application.xml	mailconnector-ear.ear/META-INF/application.xml
ejb-jar.xml	mailconnector-ear.ear/mailconnector-ejb.jar/META-INF/ejb-jar.xml
sun-ejb-jar.xml	mailconnector-ear.ear/mailconnector-ejb.jar/META-INF/sun-ejb-jar.xml
sun-web.xml	mailconnector-ear.ear/mailconnector-war.war/WEB-INF/sun-web.xml
web.xml	mailconnector-ear.ear/mailconnector-war.war/WEB-INF/web.xml

Abbildung 7.6: Das Ergebnis der Analyse der Beispielanwendung `mailconnector-ear.ear` im HTML Report.

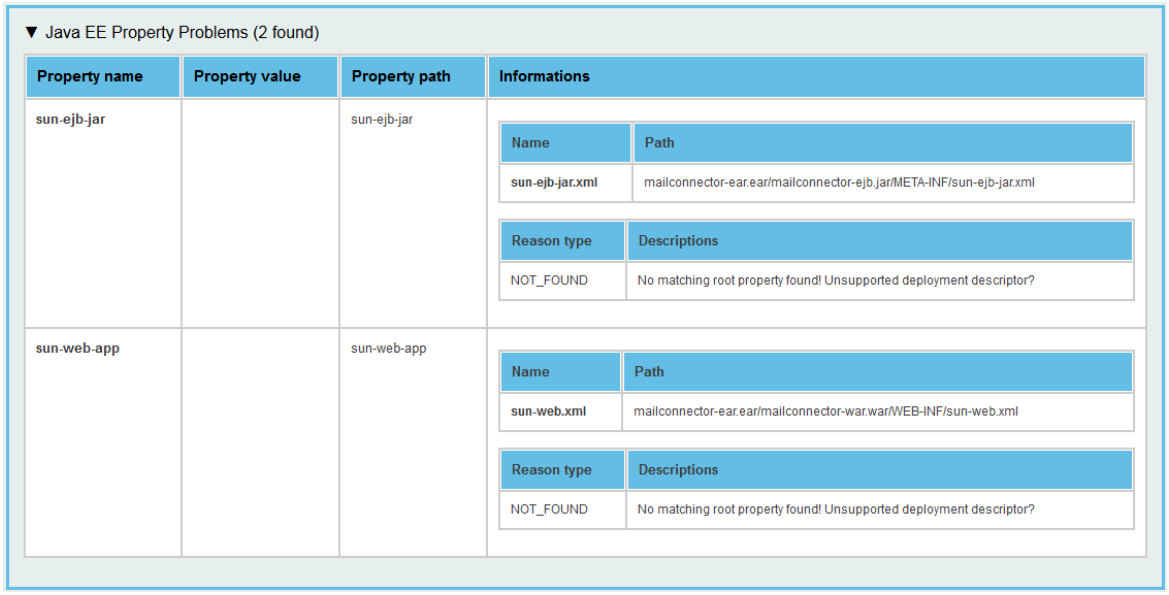
WildFly die gleichen beiden Probleme gefunden wie bei WebSphere. Da bei WebLogic der Deployment Deskriptor `sun-web.xml` verwendet werden kann, wurde dieser nicht als Problem eingestuft. Für GlassFish wurden keine Probleme mit Java EE Eigenschaften entdeckt.
Dieses Ergebnis spiegelt genau die Erwartungen wieder, die vorher für die Anwendung `mail-connector-ear.ear` aufgezeigt wurden.
In der Beispielanwendung `mailconnector.rar` waren keine Deployment Deskriptoren vorhanden, so dass auch für keinen Anwendungsserver Java EE Eigenschaftsprobleme festgestellt wurden. Bezüglich GlassFish wurden überhaupt keine Probleme festgestellt. WebLogic, WebSphere und WildFly hatten jeweils ein Classpath Problem, da die GlassFish spezifische Klas-



▼ Class path Problems (4 found)

Class name	Package	Importing classes
ConnectionSpecImpl	samples.connectors. mailconnector.share	Class name: MailBrowserServlet; Package: samples.connectors. mailconnector.servlet; Path: mailconnector-ear.ear/mailconnector-war.war/WEB-INF/classes /samples/connectors/mailconnector/servlet/MailBrowserServlet.java
JavaMailConnection	samples.connectors. mailconnector.api	Class name: MailBrowserServlet; Package: samples.connectors. mailconnector.servlet; Path: mailconnector-ear.ear/mailconnector-war.war/WEB-INF/classes /samples/connectors/mailconnector/servlet/MailBrowserServlet.java
JavaMailConnectionFactory	samples.connectors. mailconnector.api	Class name: MailBrowserServlet; Package: samples.connectors. mailconnector.servlet; Path: mailconnector-ear.ear/mailconnector-war.war/WEB-INF/classes /samples/connectors/mailconnector/servlet/MailBrowserServlet.java
JavaMailMessageListener	samples.connectors. mailconnector.api	Class name: JavaMailMessageBean; Package: samples.connectors. mailconnector.ejb.mdb; Path: mailconnector-ear.ear/mailconnector-ejb.jar/samples/connectors /mailconnector/ejb/mdb/JavaMailMessageBean.java

Abbildung 7.7: Das Ergebnis des Vergleichs der Beispielanwendung `mailconnector-ear.ear` mit dem Anwendungsserver IBM WebSphere 8.5.5.7 LIBERTY_FULL im HTML Report bezüglich gefundener Classpath Probleme.



▼ Java EE Property Problems (2 found)

Property name	Property value	Property path	Informations
sun-ejb-jar		sun-ejb-jar	Name: sun-ejb-jar.xml; Path: mailconnector-ear.ear/mailconnector-ejb.jar/META-INF/sun-ejb-jar.xml Reason type: NOT_FOUND; Descriptions: No matching root property found! Unsupported deployment descriptor?
sun-web-app		sun-web-app	Name: sun-web.xml; Path: mailconnector-ear.ear/mailconnector-war.war/WEB-INF/sun-web.xml Reason type: NOT_FOUND; Descriptions: No matching root property found! Unsupported deployment descriptor?

Abbildung 7.8: Das Ergebnis des Vergleichs der Beispielanwendung `mailconnector-ear.ear` mit dem Anwendungsserver IBM WebSphere 8.5.5.7 LIBERTY_FULL im HTML Report bezüglich gefundener Java EE Eigenschaftsprobleme.

se `org.glassfish.security.common.PrincipalImpl`, die von `samples.connectors.mailconnector.ra.inbound.MySecurityContext` verwendet wird, nicht gefunden wurde. Darüber hinaus wurden für Tomcat noch 45 weitere Classpath Probleme gefunden, die hauptsächlich zu den Java EE APIs JavaMail und JCA gehören.

Damit wurde das vorher prognostizierte Ergebnis auch für `mailconnector.rar` sehr präzise vom Prototypen bestätigt. Für GlassFish wurden bei beiden Anwendungsbeispielen keine Probleme entdeckt. Abgesehen von den beiden Deployment Deskriptoren `sun-web.xml` und `sub-ejb-jar.xml` könnte die Anwendung `mailconnector-ear.ear` zusätzlich auf WebLogic, WebSphere und WildFly installiert werden, solange der Ressourcen Adapter `mailconnector.rar` dort auch installiert ist. Tomcat als reiner Servlet Container unterstützt keine Ressourcen Adapter, EJB oder Enterprise Anwendungen, womit hier von Anfang an klar war, dass beide Beispielanwendungen nicht lauffähig sind.

7.6 Ergebnisse

In diesem Kapitel werden die gesammelten Ergebnisse und Erfahrungen des Prototypen zusammengetragen und interpretiert. Hierbei wird auch auf Probleme mit der Ontologie und eventuelle Alternativen eingegangen.

7.6.1 Zusammenfassung

Für eine Anwendung zur Analyse der Migrierbarkeit von Java EE Applikationen ist es essenziell, dass mit dem Server, für den die Anwendung programmiert wurde, keine Probleme festgestellt werden. Dies wurde in Bezug auf die beiden Beispielanwendungen aus Kapitel 7.5 und den Anwendungsserver Oracle GlassFish 3 auch gezeigt. Die gefundenen Classpath Probleme von `mailconnector-ear.ear` waren nicht relevant, da sie sich auf den ebenfalls installierten Ressourcen Adapter `mailconnector.rar` bezogen und nicht im Serverkontext zu betrachten waren.

Zusammenfassend lässt sich sagen, dass die Analyse des Classpaths und der verwendeten Java EE Eigenschaften erste Hinweise auf die Migrierbarkeit von Applikationen liefern können. Je detaillierter Applikationen und Anwendungsserver analysiert werden, umso genauer werden Probleme offengelegt und können demzufolge Probleme angegangen werden. Aus diesem Grund ist es wichtig, dass zusätzlich zu den schon analysierten Aspekten noch weitere, wie beispielsweise verwendete Methoden oder JSP-Dateien, untersucht und mit Servern abgeglichen werden.

Für eine grundlegende Einschätzung, ob die analysierte Applikation auf einem bestimmten Anwendungsserver lauffähig ist, ist der Prototyp im Wesentlichen geeignet. Jedes gefundene Problem ist dabei allerdings gesondert zu betrachten und zu gewichten. Beispielsweise schließt das Fehlen einer EJB-Implementierung auf einem Anwendungsserver die Migration einer EJB Anwendung komplett aus. Auf der anderen Seite können nicht unterstützte Deployment Deskriptoren zwar zu Warnungen führen, aber Anwendungen dennoch funktionsfähig sein.

Auch Probleme, die die Funktionsfähigkeit einer Applikation nicht beeinträchtigen, sollten dennoch gelöst und mit Äquivalenten ersetzt werden.

7.6.2 Eignungsfähigkeit einer Ontologie

In Kapitel 7.3.3.1 wurden Probleme mit der Verwendung von Regeln dargestellt. Dies betrifft die Verwendung der in Abschnitt 5.2.10 definierten Regeln in einer realen Anwendung. Alle internen Axiome, wie Klassenzuweisungen oder Gegenrelationspaare werden vom Reasoner ohne weitere Schwierigkeiten in relativ kurzer Zeit berechnet (vgl. Kapitel 7.3.3.2), solange die verwendete Ontologie nicht sehr viele Instanzen besitzt.
Zu Beginn dieser Arbeit wurde aufgrund der internen Möglichkeit Schlussfolgerungen ziehen zu können, eine Ontologie als Basis für diesen Prototypen herangezogen. Da gerade die Regeln das Ziehen von Schlüssen innerhalb der Ontologie erlaubt, muss hier rückblickend die Verwendung einer Ontologie überprüft werden.
Zuerst muss hier erwähnt werden, dass der Prototyp auf einer Ontologie aufbaut und auch funktioniert. Als Speichermedium ist eine Ontologie für diesen Einsatz sehr gut geeignet, da mit dem Reasoner einfach Abfragen durchgeführt und die Daten einfach abgespeichert und wiederverwendet werden können. Eine Erweiterung der bestehenden Ontologie um weitere Elemente, Klassen und Relationen ist sehr einfach möglich. Die internen Axiome wie Subklassen oder Gegenrelationen erleichtern den Umgang mit einer Ontologie deutlich, da beispielsweise immer nur eine Relation angelegt werden muss.
Kritikpunkt an der Ontologie ist neben dem praktischen Wegfall der internen Schlussfolgerungen, der hohe Zeitaufwand zum Starten des Reasoners bei großen Ontologien.
Die beiden hauptsächlich verwendeten Modelle, um Daten zu repräsentieren, sind Datenbanken und Ontologien. Datenbanken stellen hierbei eine Technologie dar, die spezifiziert und integriert werden muss. Zusätzlich ist jede einzelne Datenbank für genau einen Zweck entworfen worden. Das Hauptziel einer Datenbank ist es, möglichst effizient eine ganz spezielle Art von Daten zu verwalten. Ontologien stellen auf der anderen Seite ein restriktionsfreies Framework zur Verfügung, um Aspekte maschinenlesbar darzustellen. Zusätzlich können Informationen ausgetauscht und verwendet werden, um einfache Schlussfolgerungen zu ziehen. [MCBV12]
Beide Ansätze haben sowohl Vor- als auch Nachteile. Datenbanken sind tendenziell besser dafür geeignet große Datenmengen zu verwalten und können performant abgefragt werden. Ontologien können einfacher erweitert werden und vereinfachen die Verwendung von Relationen und Gegenrelationen. Welche dieser beiden Modelle verwendet wird, hängt von der Relevanz einzelner der Aspekte ab. [MCBV12]
Die Verwendung eines Reasoners erleichtert das Durchsuchen der Daten erheblich. Durch das manuelle Implementieren der relevanten Regeln in den Strukturen wurde diese Schwachstelle des Reasoners umgangen. Zusätzlich kann die Ladezeit durch das Entfernen von unwichtigen Instanzen stark reduziert werden. Daher wurde sich in dieser Arbeit für die Verwendung einer Ontologie entschieden.

8 Zusammenfassung und Ausblick

Dieses abschließende Kapitel fasst die in dieser Arbeit durchgeführten Schritte und gewonnenen Ergebnisse zusammen und gibt einen Ausblick auf nachfolgende Aufgaben und mögliche Verbesserungen.

8.1 Zusammenfassung

Alle existierenden Java EE Anwendungsserver implementieren Teile der Java EE API. Durch die unterschiedlichen Implementierungen, Herangehensweisen und Blickwinkel haben sich im Laufe der Entwicklung dieser Anwendungsserver große Differenzen zwischen ihnen gebildet. Dies schlägt sich in serverspezifischen Einstellungen und eigenen Klassen nieder, die von verschiedenen Applikationen verwendet werden können. [Ora13d]
Aus diesem Grund gibt es eine Vielzahl an Java EE Applikationen, die sich nicht direkt an den Java EE Standard halten, weil serverspezifische Implementierungen bessere Ergebnisse liefern. Sollten solche Applikationen auf andere Anwendungsserver migriert werden, weil beispielsweise keine neuen Sicherheitsupdates für den aktuellen Server vorhanden sind, so müssen die Applikationen teilweise mit sehr hohem Aufwand angepasst werden. [Sag]
Für einzelne Anwendungsserver existieren bereits Anleitungen oder Hilfstools, die notwendige Schritte für eine Migration zu diesem Anwendungsserver erleichtern sollen (Kapitel 3) [Apac; ACE+12; Oraf; Reda]. Es existiert allerdings kein allgemeines Hilfsprogramm, dass Problemstellen für einen beliebigen Anwendungsserver identifiziert oder bei der Migration unterstützt. In dieser Arbeit sollte ein solches Tool unter Zuhilfenahme einer Ontologie als Datenbasis vorgestellt werden. Ontologien werden in unterschiedlichen Szenarios als Hilfsmittel zu Migrationen und Repräsentationen eingesetzt.
Die Entscheidung für die Verwendung einer Ontologie als Datenbasis ergab sich deshalb (Kapitel 4), weil Ontologien im Gegensatz zu einfachen Datenbanken selbstständig über vordefinierte Axiome, wie Subklassen oder Gegenrelationen, Schlussfolgerungen über Instanzen ziehen können. Dies ist speziell für das Hinzufügen von Instanzen oder Relationen sehr hilfreich, da nicht jede einzelne Zuweisung oder Relation angegeben werden muss. [MCBV12]
Die Entwicklung des Datenmodells, das der hier verwendeten Ontologie zugrunde liegt, wurde nach Methoden durchgeführt, die auf Uschold und King [UK95], Grüninger und Fox [GF95] oder METHONTOLOGY [FGJ97; Góm98; FGSS99] zurückgehen. Hierfür wurden die Phasen Konzeptionalisierung (Kapitel 5), Entwicklung (Kapitel 6.1), Evaluierung (Kapitel 6.2) und Dokumentation mit den Elementen aus diesen Methoden sinnvoll befüllt und durchgeführt. Diese komplette Arbeit stellt die Dokumentation dieser Ontologie dar.
Bei der Evaluierung traten einige Probleme im Zusammenhang der Open-World Annahme der Ontologie auf. In einer Ontologie gilt eine Tatsache erst dann als falsch, wenn sie explizit so definiert ist. Aus diesem Grund konnten manche Testfragen nicht korrekt beantwortet werden. Dieses Problem konnte umgangen werden, indem für solche kritischen Abfragen

keine ontologieinternen Mechanismen verwendet wurden.
Abschließend wurde ein Prototyp entwickelt (Kapitel 7), der anhand eines Beispiels getestet wurde. Die Resultate der hier durchgeführten Analyse entsprachen den erwarteten Ergebnissen. Im Laufe der Implementierung stellte sich heraus, dass durch selbst definierte Regeln die Verarbeitung der Ontologie nicht mehr möglich war. Die benötigte Zeit für Abfragen in der Ontologie nahm quadratisch mit der Zahl der enthalten Instanzen zu. Diese beiden Probleme konnten durch manuelles Hinzufügen von wichtigen Relationen ohne Regeln und Entfernen von Elementen aus Serverontologien gelöst werden.
Auf diese Weise konnte gezeigt werden, wie eine Ontologie als Datenbasis für die Migration von Java EE Applikationen eingesetzt werden kann.

8.2 Ausblick

Die bisherige Analyse der Java EE Applikationen ist auf Classpath Elemente und Deployment Deskriptoren beschränkt. Das deckt beispielsweise nicht Annotations, Verwendung von Methoden einzelner Klassen oder JSP-Dateien ab. In einer weiterentwickelten Version des Analyseprogramms können diese Aspekte genauer untersucht werden und damit genauere Informationen zu Applikationen geliefert werden.
Durch das Analysieren von anderen Anwendungsservern können mehr Aussagen über die tatsächliche Lauffähigkeit einer Applikation geliefert werden. Auf diese Weise kann auch ausgegeben werden, auf welchem dieser Server die Anwendung am wenigsten Probleme bereitet.
Das manuelle Erstellen einer Ontologie als Eingabewert für das Programm gestaltet sich relativ schwierig, kann aber durch eine einfachere Repräsentationssprache ersetzt werden. So würde die Benutzung des Tools vereinfacht und damit der potentielle Anwenderkreis erhöht.
Die Ergebnisse werden bisher nur rudimentär präsentiert. Eine detailliertere Ansicht wie beispielsweise bei JBoss Windup würde den Informationsgehalt einzelner Probleme deutlich erhöhen. So können auch Hinweise gegeben werden, womit gefundene Probleme einfach ersetztbar wären. Damit kann die Kompatibilität zum Java EE Standard erhöht oder durch eine serverspezifische Lösung die Leistung gesteigert werden.
Eine weitere deutliche Verbesserung und auch Abgrenzung von schon vorhandenen Tools ist die Implementierung von Migrationsfunktionen. Es ist nicht möglich eine automatisierte Migration durchzuführen, allerdings sind zumindest einzelne Aspekte wie Deployment Deskriptoren auch automatisiert übersetzbar.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[AA12] Terje Aaberge und Rajendra Akerkar. „Ontology and Ontology Construction: Background and Practices.“ In: *International Journal of Computer Science and Applications* 9.2 (2012), S. 32–41. http://www.tmrfindia.org/ijcsa/v9i23.pdf.

[ABB+12] Fabio Albertoni, Tanja Baumann, Yogesh Bhatia, Euardo Monich Fronza, Marcio da Ros Gomes, Sebastian Kapciak, Catalin Mierlea, Sergio Pinto, Anoop Ramachandra, Liang Rui und Miguel Troncoso. *WebSphere Application Server V8.5. Administration and Configuration Guide for the Full Profile*. Administrationsanleitung. IBM, Juni 2012.

[ABB+13] Fabio Albertoni, Jan Bajerski, Davide Barillari, Libor Cada, Susan Hanson, Guo Liang Huang, Rispna Jain, Gabriel Knepper Mendes, Catalin Mierlea, Shishir Narain, Sergio Pinto, Jennifer Ricciuti, Carls Sadtler und Christian Steege. *WebSphere Application Server V8.5 Concepts, Planning, and Design Guide*. Konzeptanleitung. IBM, Aug. 2013.

[ACE+12] Ersan Arik, Burak Cakil, Kurtcebe Eroglu, Vasfi Gucer, Rispna Jain, Levent Kaya, Sinan Konya, Hatice Meric, Ross Pavitt, Dave Vines, Hakan Yildirim und Tayfun Yurdagul. *WebSphere Application Server V8.5 Migration Guide*. Migrationsanleitung. IBM, Nov. 2012.

[Apaa] Apache. *Apache OpenWebBeans*. http://openwebbeans.apache.org/index.html (Abgerufen am: 29. 07. 2015).

[Apab] Apache. *Apache Tomcat*. http://tomcat.apache.org/ (Abgerufen am: 29. 07. 2015).

[Apac] Apache. *Apache Tomcat - Migration Guide*. http://tomcat.apache.org/migration.html (Abgerufen am: 25. 03. 2015).

[Apad] Apache. *Apache TomEE*. http://tomee.apache.org/apache-tomee.html (Abgerufen am: 29. 07. 2015).

[Apae] Apache. *Xerces2 Java Parser 2.11.0 Release*. http://xerces.apache.org/xerces2-j/index.html (Abgerufen am: 02. 11. 2015).

[ATJ+00] Reiner Anderl, Dietmar Trippner, Harald John, Martin Arlt, Michael Endres, Jörg Katzenmaier, Martin Philipp, Christian Pütter, Alexander Angebrandt, Hans Axtner, Bernd Daum, Wilhelm Kerschbaun, Thomas Kiesewetter, Daniel Lange, Mario Leber und Konrad Pagenstert. *STEP STandard for the Exchange of Product*

Model Data: Eine Einführung in die Entwicklung, Implementierung und industrielle Nutzung der Normenreihe ISO 10303 (STEP). Vieweg+Teubner Verlag, 2000. http://books.google.de/books?id=1XeyAAAACAAJ.

[BKR+12] Raphael Barbau, Sylvere Krima, Sudarsan Rachuri, Anantha Narayanan, Xenia Fiorentini, Sebti Foufou und Ram D. Sriram. „OntoSTEP: Enriching Product Model Data Using Ontologies“. In: *Comput. Aided Des.* 44.6 (Juni 2012), S. 575–590.

[Bri] Jason Brittain. *How To Migrate Your Weblogic or WebSphere App to Tomcat*. http://blogs.mulesoft.org/how-to-migrate-your-weblogic-or-websphere-app-to-tomcat/ (Abgerufen am: 23. 02. 2015).

[BYM+08] Tim Bray, François Yergeau, Eve Maler, Jean Paoli und Michael Sperberg-McQueen. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. W3C, Nov. 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[CAB+06] Rudy Chukran, Tom Alcott, Don Bagwell, Wayne Beaton, David Dhuyvetter, Dana Duffield, Tushar Patel und Jack Snyder. *WebSphere Application Server V6 Migration Guide*. Migrationsanleitung. IBM, Aug. 2006.

[CDK+10] Rufus Credle, Dana Duffield, Vasanth Kodi, Jagdish Komakula und Anitha Krishnasmy. *WebSphere Application Server V7 Migration Guide*. Migrationsanleitung. IBM, Mai 2010.

[Chh] Bharat Chhajer. *The Story of JEE Versions*. http://www.javasprint.com/java_training_tutorial_blog/java_jee_versions_jsp_servlet_story_jsf_jstl_el_history.htm (Abgerufen am: 10. 06. 2015).

[CM13] Shing Wai Chan und Rajiv Mordani. *Java^{TM} Servlet Specification. Version 3.1*. Spezifikation. Oracle, Apr. 2013.

[Cra11] A. W. Crapo. *Semantic Application Design Language (SADL) (Version 2): A Detailed Look*. Technischer Bericht. General Electric Company, Sep. 2011. http://sadl.sourceforge.net/sadl2.html.

[CS09] Roberto Chinnici und Bill Shannon. *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v6*. Spezifikation. Sun microsystems, Dez. 2009.

[Dav] Brad Davis. *JBoss Windup & Migration Strategy*. Technischer Bericht. Red Hat. https://rhsummit.files.wordpress.com/2013/06/davis_windup-migration.pdf (Abgerufen am: 01. 05. 2015).

[DS06] Nick Drummond und Rob Shearer. *The Open World Assumption. or Sometimes its nice to know what we don't know*. Präsentation. The University of Manchester, BHIG, 2006.

[DS13a] Linda DeMichiel und Bill Shannon. *Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Web Profile Specification*. Spezifikation. Oracle, Mai 2013.

[DS13b] Linda DeMichiel und Bill Shannon. *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v7*. Spezifikation. Oracle, Mai 2013.

[Dup] Emmanuel Dupuy. *Java Decompiler – Yet another fast Java decompiler*. http://jd.benow.ca (Abgerufen am: 02. 11. 2015).

[FGJ97] Mariano Fernández-López, Asunción Gómez-Pérez und Natalia Juristo. „Methontology: from ontological art towards ontological engineering“. In: *Proc. Symposium on Ontological Engineering of AAAI*. 1997.

[FGSS99] Mariano Fernández-López, Asunción Gómez-Pérez, Juan Pazos Sierra und Alejandro Pazos Sierra. „Building a chemical ontology using Methontology and the Ontology Design Environment“. In: *IEEE Intelligent Systems* 14.5 (1999), S. 37–45.

[GB14] Ramanathan Guha und Dan Brickley. *RDF Schema 1.1*. W3C Recommendation. W3C, Feb. 2014. `http://www.w3.org/TR/2014/REC-rdf-schema-20140225/`.

[GDD09] Dragan Gasevic, Dragan Djuric und Vladan Devedzic. *Model Driven Engineering and Ontology Development*. 2nd. Springer Publishing Company, Incorporated, 2009.

[GF95] Michael Grüninger und Mark S. Fox. „Methodology for the Design and Evaluation of Ontologies“. In: 1995.

[Góm98] Asunción Gómez-Pérez. „Knowledge Sharing and Reuse“. In: *The Handbook of Applied Expert Systems*. Hrsg. von Jan Liebowitz. 1998.

[GPCFL04] Asunción Gómez-Pérez, Oscar Corcho und Mariano Fernández-López. *Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer-Verlag London Limited, 2004.

[Gru93] Thomas R. Gruber. „A Translation Approach to Portable Ontology Specifications“. In: *Knowledge Acquisition* 5.2 (Juni 1993), S. 199–220.

[GS14] Fabien Gandon und Guus Schreiber. *RDF 1.1 XML Syntax*. W3C Recommendation. W3C, Feb. 2014. `http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/`.

[HM04] Frank van Harmelen und Deborah McGuinness. *OWL Web Ontology Language Overview*. W3C Recommendation. W3C, Feb. 2004. `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof und Mike Dean. *SWRL: A Semantic Web Rule Language. Combining OWL and RuleML*. W3C Member Submission. W3C, Mai 2004. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`.

[IBMa] IBM. *About WebSphere Liberty*. `https://developer.ibm.com/wasdev/websphere-liberty/` (Abgerufen am: 01. 05. 2015).

[IBMb] IBM. *IBM WebSphere Application Server Migration Toolkit*. `http://www.ibm.com/developerworks/websphere/downloads/migtoolkit/compmig.html` (Abgerufen am: 23. 02. 2015).

[IBMc] IBM. *WebSphere*. `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/websphere/` (Abgerufen am: 08. 07. 2015).

[IBM13] IBM. *WebSphere Application Server Family*. Datenblatt. IBM, Sep. 2013.

[Iee] „IEEE Standard for Developing Software Life Cycle Processes“. In: *IEEE Std 1074-1995* (1996), S. i–.

[Jam10] Lisa Jamen. *Oracle®Fusion Middleware. 11g Release 1 (11.1.1)*. Konzeptanleitung. Oracle, 2010.

[KCM14] Graham Klyne, Jeremy J. Carroll und Brian McBride. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[Kin] Cheryl King. *What is the Migration Toolkit for Application Binaries (Tech Preview)?* https://developer.ibm.com/wasdev/docs/migration-toolkit-application-binaries-tech-preview/ (Abgerufen am: 01. 05. 2015).

[KKS14] Amarjit Kaur, Usvir Kaur und Dr. Dheerendra Singh. „Server Migration Ontology Compatibility for Hybrid Layer Data Transfer: A Review“. In: *International Journal of Computing and Technology* 1.6 (2014), S. 242–244. http://www.ijcat.org/IJCAT-2014/1-6/Server-Migration-Ontology-Compatibility-for-Hybrid-Layer-Data-Transfer-A-Review.pdf.

[KMH+] Gavin King, Pete Muir, Jozef Hartinger, Dan Allen und David Allen. *Weld 2.2.14.Final - CDI Reference Implementation. CDI: Contexts and Dependency Injection for the Java EE platform*. Referenzanleitung. Red Hat.

[KR13a] Mohanned M. Kazzaz und Marek Rychlý. „A Web Service Migration Framework“. In: *Proceedings of the Eighth International Conference on Internet and Web Applcations and Services*. 2013, S. 58–63.

[KR13b] Mohanned M. Kazzaz und Marek Rychlý. „Web Service Migration with Migration Decisions Based on Ontology Reasoning“. In: *Proceedings of the Twelfth International Conference on Informatics - Informatics'2013*. Faculty of Electrical Engineering und Informatics, University of Technology Košice, 2013, S. 186–191. http://www.fit.vutbr.cz/research/view_pub.php?id=10434.

[KS08] Guido Krüger und Thomas Stark. *Handbuch der Java Programmierung. Standard Edition Version 6*. Addison-Wesley, 2008.

[LGJ+98] Jintae Lee, Michael Grüninger, Yan Jin, Thomas Malone, Austin Tate und Gregg Yost. „PIF The Process Interchange Format“. In: *Handbook on Architectures of Information Systems*. Hrsg. von Peter Bernus, Kai Mertins und Günter Schmidt. International Handbooks on Information Systems. Springer Berlin Heidelberg, 1998, S. 167–189.

[Ltd] Thai Open Source Software Center Ltd. *Trang – Multi-format schema converter based on RELAX NG*. http://www.thaiopensource.com/relaxng/trang.html (Abgerufen am: 02. 11. 2015).

[MB12] N. Madurai Meenachi und M. Sai Baba. „Web Ontology Language Editors for Semantic Web – A Survey“. In: *International Journal of Computer Applications* 53.12 (Sep. 2012), S. 12–16.

[MCBV12] Carmen Martinez-Cruz, Ignacia J. Blanco und M. Amparo Vila. „Ontologies Versus Relational Databases: Are They So Different? A Comparison“. In: *Artif. Intell. Rev.* 38.4 (Dez. 2012), S. 271–290.

[Met] Metawerx. *Metawerx Wiki*. http://wiki.metawerx.net/ (Abgerufen am: 09. 06. 2015).

[NM01] Natalya F. Noy und Deborah L. McGuinness. *Ontology Development 101: A Guide to Creating Your First Ontology*. Vorlesungsskript. Stanford University – Knowledge Systems Laboratory, 2001.

[NR09] Toby Nixon und Alain Regnier. *Web Services Dynamic Discovery (WS-Discovery) Version 1.1*. OASIS Standard. OASIS Open, Juli 2009. http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf.

[Oraa] Oracle. *GlassFish Logos*. https://glassfish.java.net/glassfish_buttons/ (Abgerufen am: 02. 07. 2015).

[Orab] Oracle. *Java EE and GlassFish Server Roadmap Update*. https://blogs.oracle.com/theaquarium/entry/java_ee_and_glassfish_server (Abgerufen am: 01. 07. 2015).

[Orac] Oracle. *Java EE Code Samples & Apps*. http://www.oracle.com/technetwork/java/javaee/documentation/code-139018.html (Abgerufen am: 06. 11. 2015).

[Orad] Oracle. *Java EE: XML Schemas for Java EE Deployment Descriptors*. http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html (Abgerufen am: 20. 06. 2015).

[Orae] Oracle. *M2GMigrationGuide*. https://wikis.oracle.com/display/GlassFish/M2GMigrationGuide (Abgerufen am: 25. 02. 2015).

[Oraf] Oracle. *Performing a Side-By-Side Upgrade With Upgrade Tool*. http://docs.oracle.com/cd/E18930_01/html/821-2437/abmbr.html#gdkix (Abgerufen am: 01. 05. 2015).

[Orag] Oracle. *Sun Java System Application Server Platform Edition 9 Upgrade and Migration Guide*. https://docs.oracle.com/cd/E19501-01/819-3663/abmcc/index.html (Abgerufen am: 16. 01. 2015).

[Ora10] Oracle. *Oracle GlassFish Server. Frequently Asked Questions*. FAQ. Oracle, 2010. http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server-faq-071872.pdf.

[Ora13a] Oracle. *GlassFish Server Open Source Edition. Release Notes, Release 4.0*. Release Notes. Oracle, Mai 2013.

[Ora13b] Oracle. *GlassFish Server Open Source Edition. Application Deployment Guide, Release 4.0*. Anleitung. Oracle, Mai 2013.

[Ora13c] Oracle. *Introducing Oracle WebLogic Server 12c, Release 12.1.2. The #1 Application Server across Conventional and Cloud Environments*. Einführung. Oracle, Mai 2013.

[Ora13d] Oracle. *Introduction to Java Platform, Enterprise Edition 7*. Oracle White Paper. Oracle, Juni 2013.

[Ora14a] Oracle. *Java Platform, Enterprise Edition. The Java EE Tutorial, Release 7*. Anleitung. Oracle, Sep. 2014.

[Ora14b] Oracle. *Oracle WebLogic Server*. Datenblatt. Oracle, 2014.

[Ora14c] Oracle. *Oracle WebLogic Server 12.1.3. Developing with WebLogic Server*. Oracle White Paper. Oracle, Juni 2014.

[Ora14d] Oracle. *Oracle®Fusion Middleware, Developing Oracle Coherence Applications for Oracle WebLogic Server. 12c (12.1.3)*. Entwicklungsanleitung. Oracle, Juni 2014.

[Ora15a] Oracle. *Oracle®Fusion Middleware, Administering JDBC Data Sources for Oracle WebLogic Server. 12c (12.1.3)*. Administrationsanleitung. Oracle, März 2015.

[Ora15b] Oracle. *Oracle®Fusion Middleware, Administering JMS Resources for Oracle WebLogic Server. 12c (12.1.3)*. Administrationsanleitung. Oracle, Jan. 2015.

[Ora15c] Oracle. *Oracle®Fusion Middleware, Configuring and Using the Diagnostics Framework for Oracle WebLogic Server. 12c (12.1.3)*. Handbuch. Oracle, Jan. 2015.

[Ora15d] Oracle. *Oracle®Fusion Middleware, Deploying Applications to Oracle WebLogic Server. 12c (12.1.3)*. Deploymentanleitung. Oracle, Feb. 2015.

[Ora15e] Oracle. *Oracle®Fusion Middleware, Developing Applications for Oracle WebLogic Server. 12c (12.1.3)*. Entwicklungsanleitung. Oracle, März 2015.

[Ora15f] Oracle. *Oracle®Fusion Middleware, Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server. 12c (12.1.3)*. Entwicklungsanleitung. Oracle, Apr. 2015.

[RB] Jim des Rivieres und Wayne Beaton. *Eclipse Platform Technical Overview*. `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html` (Abgerufen am: 02. 11. 2015).

[Reda] Red Hat. *What ist Windup*. `http://people.redhat.com/~pmuir/windup/documentation.html` (Abgerufen am: 03. 05. 2015).

[Redb] Red Hat. *Windup Migration Platform*. `http://windup.jboss.org/` (Abgerufen am: 25. 02. 2015).

[Redc] Red Hat. *JBoss Application Server 7 > JBoss AS History*. `http://jbossas.jboss.org/history` (Abgerufen am: 09. 07. 2015).

[Redd] Red Hat. *WildFly Logos*. `http://design.jboss.org/wildfly/` (Abgerufen am: 09. 07. 2015).

[Rede] Red Hat. *Admin Guide*. Administrationsanleitung. PDF Export als `wildfly9-admin-guide.pdf`. `https://docs.jboss.org/author/display/WFLY9/Admin+Guide` (Abgerufen am: 09. 07. 2015).

[Redf] Red Hat. *Developer Guide*. Entwicklungsanleitung. PDF Export als `wildfly 8-devel-guide.pdf`. `https://docs.jboss.org/author/display/WFLY9/Developer+Guide#` (Abgerufen am: 09. 07. 2015).

[Redg] Red Hat. *JBossTS 4.14.0. Server Integration Guide*. Serverintegrationsanleitung. Red Hat, 2010.

[Redh] Red Hat. *IronJacamar 1.2 User's Guide. Connecting your Enterprise Information Systems*. Benutzeranleitung. Red Hat.

[Redi] Red Hat. *RESTEasy JAX-RS. RESTFul Web Services for Java, 3.0.9.Final*. Referenzanleitung. Red Hat.

[Rom] Lukasz Romaszewski. *Migrate to Oracle WebLogic Server 11g*. `http://de.slideshare.net/oracle_imc_team/partner-webcast-migration-to-weblogic-server-11g` (Abgerufen am: 25. 02. 2015).

[ron] ronmamo. *Java runtime metadata analysis, in the spirit of Scannotations*. `https://github.com/ronmamo/reflections` (Abgerufen am: 02. 11. 2015).

[Sag] Ajit Sagar. *Application Server Migration Presentation*. `http://de.slideshare.net/Aamir97/application-server-migration-presentation` (Abgerufen am: 25. 02. 2015).

[Sal14] Alexander Salvanos. *Professionell entwickeln mit Java EE 7. Das umfassende Handbuch*. Bonn: Galileo Press, 2014.

[SFGPMG12] Mari-Carmen Suárez-Figueroa, Asunción Gómez-Pérez, Enrico Motta und Aldo Gangemi. *Ontology Engineering in a Networked World*. Berlin: Springer, 2012.

[Sha06] Bill Shannon. *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v5*. Spezifikation. Sun microsystems, Mai 2006.

[SMMS02] Nenad Stojanovic, Alexander Maedche, Boris Motik und Nenad Stojanovic. „User-driven ontology evolution management". In: *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW'02*. Madrid, 2002.

[Sou] Sourceforge.net. *The OWL API*. `http://owlapi.sourceforge.net/` (Abgerufen am: 02. 11. 2015).

[SS09] Steffen Staab und Rudi Studer. *Handbook on Ontologies*. 2nd. Springer Publishing Company, Incorporated, 2009.

[STF+] Clebert Suconic, Andy Taylor, Tim Fox, Jeff Mesnil und Howard Gao. *HornetQ QuickStart Guide. Putting the buzz in messaging*. Referenzanleitung. Red Hat.

[Stu11] H. Stuckenschmidt. *Ontologien: Konzepte, Technologien und Anwendungen - 2. Auflage*. Informatik im Fokus. Springer, 2011. `http://books.google.de/books?id=ndwfBAAAQBAJ`.

[Sun09] Sun. *Java^{TM} EE Connector Architecture Specification. Version 1.6*. Spezifikation. Sun microsystems, Nov. 2009.

[SVS12] R SureshKumar, V. Vijayakaran und T. T. SampathKumar. „Ontology based Secured Data Migration“. In: *International Journal of Modern Engineering Research (IJMER)* 2 (3 2012), S. 1343–1347. http://www.ijmer.com/papers/vol2_issue3/EI2313431347.pdf.

[SWA+94] Guus Schreiber, Bob Wielinga, Hans Akkermans, Walter Van de Velde und Anjo Anjewierden. „CML: the commonKADS conceptual modelling language“. In: *A Future for Knowledge Acquisition*. Bd. 867. Lecture Notes in Computer Science. Berlin, Germany: Springer, 1994, S. 1–25. http://doc.utwente.nl/83031/.

[SWJ95] Guus Schreiber, Bob Wielinga und Wouter Jansweijer. „The KACTUS View on the 'O' Word“. In: *The 1995 International Joint Conference on AI: Montreal, Quebec, Canada: 1995, August, 20-25*. Hrsg. von D. Skuce. Workshop on Basic Ontological Issues in Knowledge Sharing. 1995, S. 15.1–15.10.

[UG96] Mike Uschold und Michael Grüninger. „Ontologies: Principles, methods and applications“. In: *Knowledge Engineering Review* 11 (1996), S. 93–136.

[UJ99] Mike Uschold und Robert Jasper. „A framework for understanding and classifying ontology applications“. In: *Proceedings of the IJCAI99 Workshop on Ontologies*. 1999, S. 16–21. http://www.uni-leipzig.de/~tbittner/courses/GEOID/uschold99FrameworkUnderstandingOntology.pdf.

[UK95] Mike Uschold und Martin King. „Towards a Methodology for Building Ontologies“. In: *In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95*. 1995.

[Vat13] Marina Vatkina. *JSR 345: Enterprise JavaBeans^{TM}, Version 3.2. EJB Core Contracts and Requirements*. Spezifikation. Oracle, Apr. 2013.

[VG11] Aleksa Vukotic und James Goodwill. *Apache Tomcat 7*. apress, 2011.

[Wat85] Donald Arthur Waterman. *A Guide to Expert Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1985.

[Web] „Comparing IBM WebSphere and Oracle WebLogic. Benefits of an IBM WebSphere application infrastructure“. In: (Nov. 2011).

[YSD+03] Seong Steve Yu, Barry Searle, Dana Duffield, Wayne Beaton, Tom Alcott, Radhika Iyer, David Dhuyvetter, Sharad Cocasse und Jack Snyder. *Migrating to WebSphere V5.0. An End-to-End Migration Guide*. Migrationsanleitung. IBM, Apr. 2003.