Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Studienarbeit Nr. 2410

Vorlagen für das Deployment von Services und Applikationen in der Cloud

Shaojun Zhang

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Johannes Wettinger
begonnen am:	30.11.2012
beendet am:	25.04.2013
CR-Klassifikation:	K6; D.2.11; D2.13

Kurzfassung

Aktuell werden unterschiedliche Verwaltungswerkzeuge wie Juju [1] oder Chef [2] [23] im Bereich des Cloud Computing entwickelt um das Deployment und die Verwaltung von Services und Applikationen in der Cloud zu erleichtern. Mit Hilfe dieser Werkzeuge können Artefakte entwickelt und verwaltet werden, um die automatisierte Installation und Konfiguration von Softwarekomponenten zu ermöglichen. Diese Artefakte können miteinander kombiniert werden um Vorlagen für Cloud-Services („Service-Templates“) zu erstellen, die sich aus mehreren Softwarekomponenten zusammensetzen. Das Hauptproblem ist hierbei, dass die Artefakte nicht portabel sind weil sie durch proprietäre Ansätze implementiert werden und damit von einer ebenfalls proprietären Laufzeitumgebung abhängig sind.

Um das oben genannte Problem zu vermeiden sind Standardisierungsbemühungen im Bereich des Cloud Computing von wichtiger Bedeutung. Die "Topology and Orchestration Specification for Cloud Applications" (TOSCA) [3] stellt einen Standardisierungsansatz dar, um die Portabilität von Cloud-Services und der zugrundeliegenden Vorlagen und Artefakte zu verbessern [10].

Ziel dieser Studienarbeit ist der Entwurf und die Entwicklung einer Prozedur, mit der existierende Artefakte aus der Juju-Community zu TOSCA-konformen Artefakte konvertiert werden können. Damit können diese Artefakte und entspr. Vorlagen, die diese Artefakte verwenden, von jeder TOSCA-konformen Laufzeitumgebung verarbeitet werden.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Einführung 1

1.2 Aufgabenstellung 1

1.3 Struktur der Arbeit 1

2 Grundlagen 3

2.1 Juju 3

2.1.1 Juju Charm 3

2.1.2 Das Verzeichnis "hooks" 4

2.1.3 Die Datei "metadata.yaml" 5

2.1.4 Die Datei "config.yaml" 6

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA) 7

2.2.1 TOSCA-Kernbegriffe 7

2.2.2 TOSCA Cloud Service Archive (CSAR) 9

2.2.3 TOSCA-Definitions Dokument 10

2.2.3.1 Definitions 12

2.2.3.2 Import 12

2.2.3.3 Requirement Types 12

2.2.3.4 Capability Types 13

2.2.3.5 Artifact Types 13

2.2.3.6 Artifact Templates 14

2.2.3.7 Node Types 15

2.2.3.8 Node Type Implementations 16

3 Entwurf 18

3.1 Analyse der Prozedur 18

3.2 Konzept der Prozedur 19

3.3 Entwurf der zu implementirenden Pakete 20

3.3.1 TOSCA-CSAR-Generator....21
3.3.2 ZIP-Bearbeiter....21
3.3.3 Juju-Yaml-Reader....21
3.3.4 TOSCA-XML-Generator....21
3.4 Verwendetes Java-Paket und Software von Drittanbietern....21
3.4.1 JDK....21
3.4.2 SnakeYaml....21
3.4.3 Dom4j....22
3.5 Komplettes Sequenzdiagramm....22
4 Implementierung....24
4.1 Implementirung des Pakets "org.tosca"....24
4.1.1 Klasse "CharmToNodeType"....24
4.1.1.1 Methode "main"....24
4.1.1.2 Klasse "Transform"....24
4.2 Implementierung des Pakets "org.tosca.zip"....24
4.2.1 Klasse "FileModel"....24
4.2.2 Klasse "FileModelList"....25
4.2.2.1 Methode "format"....25
4.2.3 Klasse "ZipUtil"....25
4.2.3.1 Methode "getAllFilenames"....26
4.2.3.2 Methode "getFileFromZip"....26
4.2.3.3 Methode "format"....27
4.2.3.4 Methode "addFileToZip"....28
4.3 Implementierung des Pakets "org.tosca.yaml"....28
4.3.1 Klasse "YamlModel"....28
4.3.2 Klasse "YamlModelList"....29

4.3.3 Klasse "YamlReader" 29
4.3.3.1 Methode "readYamlFile" 29
4.3.3.2 Methode "read" 30
4.4 Implementierung des Pakets "org.tosca.xml" 30
4.4.1 Klasse "XmlElementsModel" 30
4.4.2 Klasse "XmlElementsImpl" 30
4.4.2.1 Methode "elementsImpl" 31
4.4.2.2 Methode "rootImpl" 32
4.4.2.3 Methode "importImpl" 32
4.4.2.4 Methode "requirementTypeImpl" 33
4.4.2.5 Methode "capabilityTypeImpl" 33
4.4.2.6 Methode "artifactTypeImpl" 33
4.4.2.7 Methode "artifactTemplateImpl" 33
4.4.2.8 Methode "nodeTypeImpl" 33
4.4.2.9 Methode "nodeTypeImplementationImpl" 34
4.4.3 Klasse "XMLGenerator" 34
4.4.3.1 Methode "generator" 34
4.4.4 Klasse "XsdElementsModel" 34
4.4.5 Klasse "XsdElementsImpl" 34
4.4.5.1 Klasse "ConfigModel" 35
4.4.5.2 Methode "elementsImpl" 35
4.4.6 Klasse "XSDGenerator" 36
4.4.6.1 Methode "generator" 36
5 Zusammenfassung und Ausblick 37
Literaturverzeichnis 38
Erklärung 39

Abbildungsverzeichnis

Abbildung 2.1: Ein Beispiel für die Struktur eines Charm..... 3

Abbildung 2.2: Ein Beispiel für ein "hooks" Verzeichniss..... 4

Abbildung 2.3: Eine "metadata.yaml" Datei des Charm "drupal" [6]..... 5

Abbildung 2.4: Eine "metadata.yaml" Datei des Charm "mysql" [6].....6

Abbildung 2.5: Eine "config.yaml" Datei des Charm "myblog" [7]..... 7

Abbildung 2.6: Strukturelle Elemente eines Service-Template und ihrer Beziehungen [8].. 8

Abbildung 2.7: Die Struktur einer CSAR-Datei..... 10

Abbildung 3.1: Zusammenhang zwischen der Eingabe und der Ausgabe der Prozedur..... 18

Abbildung 3.2: Ein einfaches Pseudo-Sequenzdiagramm der Prozedur.....20

Abbildung 3.3: Ein Sequenzdiagramm der Prozedur.....23

Abbildung 4.1: Das Speichern von Informationen in einer "metadata.yaml" Datei.....29

Abbildung 4.2: Das Speichern von Informationen in einer "config.yaml" Datei.....35

Ausschnittsverzeichnis

Ausschnitt 2.1: XML-Syntax eines TOSCA-Definitions-Dokuments.....11

Ausschnitt 2.2: XML-Schema für ein Node-Type-Properties-Dokument..... 15

Tabellenverzeichnis

Tabelle 4.1: Parameter der Methode "format" 25

Tabelle 4.2: Parameter der Methode "getAllFilenames" 26

Tabelle 4.3: Parameter der Methode "getFileFromZip" 26

Tabelle 4.4: Parameter der Methode "format" 27

Tabelle 4.5: Parameter der Methode "addFileToZip" 28

Tabelle 4.6: Parameter der Methode "elementsImpl" 31

Abkürzungsverzeichnis

API: Application Program Interface

BPEL: Business Process Execution Language

BPMN: Business Process Model and Notation

CMS: Content Management System

CSAR: Cloud Service Archive

JDK: Java Development Kit

REST: Representational State Transfer

TOSCA: Topology and Orchestration Specification for Cloud Applications

URI: Uniform Resource Identifier

UUID: Universal Unique Identifier

WSDL: Web Services Description Language

XML: Extensible Markup Language

XSD: Extensible Schema Definition

1 Einleitung

1.1 Einführung

Unter Verwendung von Konfigurationsverwaltungs- und Orchestrierungswerkzeuge können zur Zeit Anwendungen und Services in der Cloud bereitgestellt werden. Diese Werkzeuge ermöglichen die Bereitstellung von virtuellen Maschinen sowie die Installation und die Konfiguration von Softwarekomponenten auf virtuellen Maschinen. Einer der Vorteile dieser Werkzeuge ist: Entwickler veröffentlichen wiederverwendbare Artefakte ("Service-Templates") zur Installation und Konfiguration von Softwarekomponenten wie Apache-Web-Server oder MySQL-Datenbankserver in der Cloud. Der Nachteil dieser Artefakte ist jedoch die Tatsache, dass sie durch proprietäre Ansätze implementiert werden. Folglich können die Artefakte nur durch das spezielle Werkzeug, mit dem sie erstellt werden, verarbeitet werden. Das bedeutet, dass die Artefakte nicht portabel sind. Um dieses Problem zu lösen, wird aktuell ein Standard entworfen, der die Portabilität solcher Artefakte verbessern soll: Topology and Orchestraton Specification for Cloud Applications (TOSCA) [3]. TOSCA befindet sich noch in der Entwicklung und bezweckt die Erstellung von portablen Service-Templates, damit jede TOSCA-Laufzeitumgebung diese Service-Templates verarbeiten kann [10].

1.2 Aufgabenstellung

Das Ziel der vorliegenden Arbeit ist, eine Prozedur zum Erstellen von TOSCA-Service-Templates zu entwerfen und zu implementieren. Diese Service-Templates basieren auf vorhandenen Artefakten, die durch die Juju-Community [4] veröffentlicht und zur Verfügung gestellt werden.

Das Hauptziel dieser Arbeit ist, eine Prozedur zum Generieren von TOSCA-Node-Types zu entwickeln. Die Prozedur implementiert eine Konvertierung von Juju-Charms zu TOSCA-Node-Types. Node-Types sind ein wichtiger Teil eines Service-Template und werden in einem TOSCA-Definitions-Dokument definiert.

Die Details zum TOSCA-Definitions-Dokument sowie seine Elemente *ServiceTemplate*, *NodeType* etc. werden in dem Grundlagenkapitel (Unterkapitel 2.2) vorgestellt.

1.3 Struktur der Arbeit

Die Arbeit ist in mehrere Kapitel gegliedert. Auf die Einleitung folgt ein Grundlagenkapitel. In diesem Grundlagenkapitel werden existierende Technologien und Lösungen wie Juju und TOSCA beschrieben, die als Grundlage dieser Arbeit dienen.

Im dritten Kapitel wird der Entwurf der Anwendungsarchitektur präsentiert. Die Unterkapitel beschreiben einige Softwarekomponenten, die zur Implementierung benötigt werden.

Das vierte Kapitel zeigt die Implementierung des Entwurfs. Damit soll gezeigt werden, dass die Aufgabenstellung tatsächlich realisiert werden kann.

Kapitel 5 beinhaltet eine Zusammenfassung der Arbeit und ein kurzer Ausblick auf mögliche weiterführende Arbeiten.

2 Grundlagen

Aus der Aufgabenstellung (Unterkapitel 1.2) ist die Hauptfunktion der umzusetzenden Prozedur bekannt, die ein Artefakt von Juju einliest und daraus ein TOSCA-Node-Type erzeugt. Im Folgenden werden Eingabe und Ausgabe der umgesetzten Prozedur beschrieben. Dabei werden auch grundlegende Informationen zu Juju und TOSCA zur Verfügung gestellt.

2.1 Juju

Juju zielt darauf ab, ein Service-Deployment- und Orchestrierungswerkzeug zu sein, das die Zusammenarbeit zwischen den Services sowie die einfache Verwaltung dieser Services ermöglicht. Verschiedene Service-Entwickler können mit Juju Services selbständig erstellen und die Kommunikation von diesen Services durch ein einfaches Konfigurationsprotokoll koordinieren. Dann können die Service-Benutzer die Services von verschiedenen Service-Entwicklern nehmen und sie sehr komfortabel in einer Umgebung bereitstellen. Das Ergebnis ist, dass mehrere Maschinen und Komponenten transparent zusammenarbeiten können, um die angeforderten Services zur Verfügung zu stellen.

2.1.1 Juju Charm

Die Eingabe der Prozedur ist eine ZIP-Datei, die von der Juju-Community veröffentlicht und als "Charm" bezeichnet wird. Charms definieren, wie sich Services integrieren und wie ihre Service-Einheiten auf Ereignisse in der verteilten Umgebung reagieren. Eine Service-Instanz in Juju besitzt zu Beginn genau eine Service-Einheit. Es können jedoch weitere Service-Einheiten zu dieser Instanz hinzugefügt werden, um z.B. Skalierbarkeit zu ermöglichen. Bspw. kann eine MySQL-Datenbank-Instanz zu Beginn genau eine Service-Einheit besitzen (eine virtuelle Maschine). Später können dann weitere Service-Einheiten (weitere virtuelle Maschinen) zu dieser Instanz hinzugefügt werden und mit der ursprünglichen Service-Einheit verknüpft werden. Ein Charm stellt die Definition des Service zur Verfügung. Zur Definition gehören auch seine Metadaten, die Abhängigkeiten von anderen Services, die notwendigen Pakete sowie die Verwaltung der Anwendung. In Abbildung 2.1 wird ein Beispiel für die Struktur eines Charm dargestellt. "xxx" ist der Namen eines beliebigen Service.

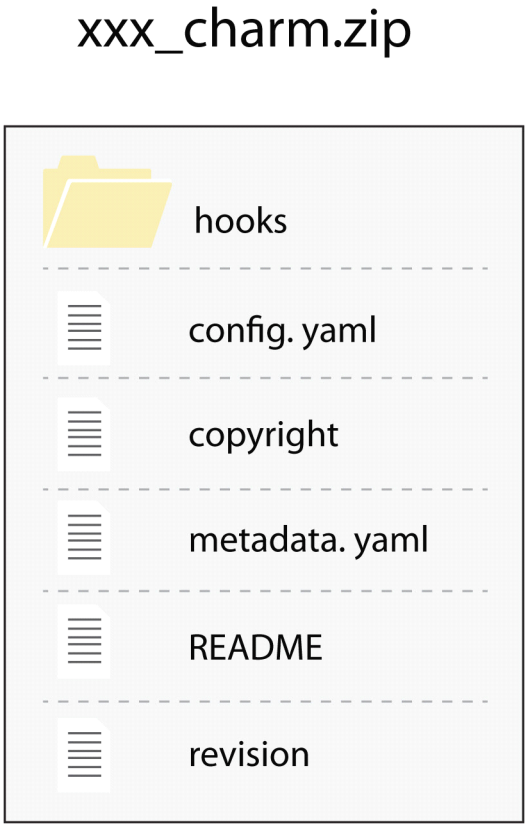

Abbildung 2.1: Ein Beispiel für die Struktur eines Charm

Normalerweise enthält jedes Charm eine "metadata.yaml" Datei und ein Verzeichnis namens "hooks". Manche Charms enthalten noch eine "config.yaml" Datei.

2.1.2 Das Verzeichnis "hooks"

In dem Verzeichnis "hooks" gibt es viele Dateien. Jede dieser Dateien wird als "Hook" bezeichnet. Die Hooks in einem Charm sind ausführbare Dateien, die unter Verwendung von einer beliebigen Skript-Sprache oder Programmiersprache geschrieben werden können. Juju verwendet die Hooks, um eine Service-Einheit über die Veränderungen in ihrem Lebenszyklus oder in der verteilten Umgebung zu benachrichtigen. Ein für eine Service-Einheit laufendes Hook kann diese Umgebung überprüfen. Außerdem kann es die gewünschten lokalen Änderungen auf der Maschine, wo sich dieses Hook befindet, vornehmen sowie die Einstellung der Relation ändern.

In der Regel gibt es in Bezug auf den Lebenszyklus einer Service-Einheit folgende Hooks: "install", "start" und "stop" [5]. Es kann noch weitere Hooks geben, die als "Relation-Hook" bezeichnet werden. Sie werden auf jeder Service-Einheit aufgerufen, wenn eine Relation hergestellt oder geändert wird. Ein Beispiel für ein "hooks" Verzeichnis wird in Abbildung 2.2 gezeigt. In diesem Verzeichnis "hooks" gibt es zwei Relation-Hooks "db-relation-joined" und "db-relation-broken". Das Hook "db-relation-joined" wird aufgerufen, wenn eine Relation - z.B. eine Datenbankverbindung - zu einer Service-Einheit hinzugefügt wird. Das Hook "db-relation-broken" wird aufgerufen, wenn die Relation entfernt wird. Dabei wird die Service-Einheit die Konfigurationsinformationen zur Datenbankverbindung löschen.

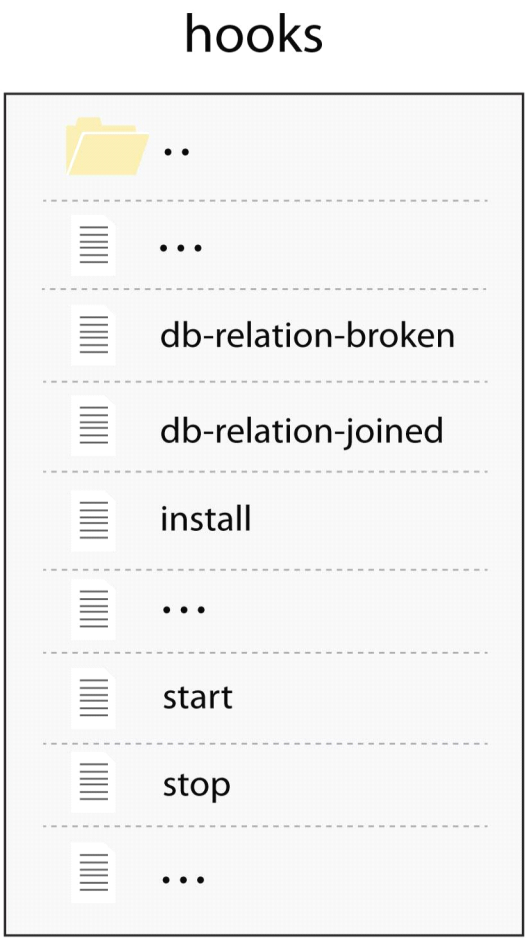

Abbildung 2.2: Ein Bespiel für ein "hooks" Verzeichnis

2.1.3 Die Datei "metadata.yaml"

YAML [28] ist eine einfache Markup-Sprache zur Datenserialisierung, die sowohl gut von Menschen lesbar sein soll als auch vollautomatisch von Maschinen verarbeitbar ist. Die Datei "metadata.yaml", die sich im Wurzelverzeichnis eines Charm befindet, beschreibt das Charm und enthält die Metadaten für das Charm. Wir nehmen das Charm "drupal" für das Deployment des CMS-System als Beispiel [29]. Seine "metadata.yaml" Datei wird in Abbildung 2.3 dargestellt.

```
name: drupal
summary: "Drupal CMS"
maintainer: "Drupal PowerUser <drupaluser@somedomain.foo>"
description: |
   Installs the drupal CMS system, relates to the mysql charm provided in
   examples directory. Can be scaled to multiple web servers
requires:
  db:
   interface: mysql
```

Abbildung 2.3: Eine "metadata.yaml" Datei des Charm "drupal" [6]

Diese Datei "metadata.yaml" deklariert ein Charm mit dem Namen "drupal". Die ersten vier Abschnitte geben folgende Informationen über dieses Charm an: den Namen des Charm, die Information über den Ersteller des Charm, eine kurze und eine lange Beschreibung. Der letzte Abschnitt ist "requires". Dies beschreibt einen Interface-Typ, der von diesem Charm benötigt wird. Da das Charm "drupal" eine MySQL-Datenbank als Service benötigt, muss dies in den Metadaten angegeben werden. Da dieses Charm keinen Service für andere Charms zur Verfügung stellt, gibt es keinen "provides" Abschnitt. Was bedeutet das Interface "mysql"? Die Antwort ist in der Interface-Information aus der "metadata.yaml" Datei des Charm namens "mysql" zu finden. Ein Beispiel der Datei "metadata.yaml" des Charm "mysql" wird in Abbildung 2.4 gezeigt.

```
name: mysql
summary: "MySQL relational database provider"
maintainer: "Joe Charmer <youremail@whatever.com>"
description: |
  Installs and configures the MySQL package (mysqldb), then runs it.

  Upon a consuming service establishing a relation, creates a new
  database for that service, if the database does not yet
  exist. Publishes the following relation settings for consuming
  services:

    database: database name
    user: user name to access database
    password: password to access the database
    host: local hostname
provides:
 db:
  interface: mysql
```

Abbildung 2.4: Eine "metadata.yaml" Datei des Charm "mysql" [6]

In der letzten Zeile ist das Interface erkennbar, welches uns vom Charm "mysql" zur Verfügung gestellt wird.

2.1.4 Die Datei "config.yaml"

Die Datei "config.yaml" befindet sich auch im Wurzelverzeichnis eines Charm. In dieser Datei werden einige Konfigurationsoptionen definiert, auf die das Charm zugreift. Charms erlauben nur, die Konfigurationsoptionen zu bearbeiten, die von dem Ersteller des Charm bekannt gegeben werden. Diese Optionen werden nicht nur für eine bestimmte Service-Einheit oder Beziehung verwendet, sondern für den gesamten Service. Beispielsweise definiert der Service "myblog" eine "blog-title" Option. Diese Option kontrolliert den Titel des zu veröffentlichenden Blogs. Die Änderungen an dieser Option gelten für alle Service-Einheiten, die zu einer bestimmten Service-Instanz des Service "myblog" gehören. Dabei wird ein Hook auf jeder von diesen Service-Einheiten aufgerufen.

```
options:
  port:
    default: 80
    type: int
    description: Port to listen on
  admin-email:
    # type: str is implied
    default: null
    description: Email address for the site administrator.
```

Abbildung 2.5: Eine "config.yaml" Datei des Charm "myblog" [7]

In Abbildung 2.5 wird gezeigt, wie eine "config.yaml" Datei aussieht. Die Information enthält eine lesbare Beschreibung und einen optionalen Default-Wert "default". Zusätzlich kann möglicherweise ein Typ "type" spezifiziert werden. Alle Optionen haben einen Default-Typ von 'string'. Er bedeutet, dass sein Wert nur als eine Text-Zeichenfolge behandelt wird. Andere gültige Optionen sind 'int' und 'float'.

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Cloud Computing [8] [9] kann wertvoller werden, wenn die (semi-)automatische Erstellung und Verwaltung von Services auf der Anwendungsschicht in den verschiedenen Cloud-Umgebungen eingesetzt werden kann. Somit können die Services interoperabel bleiben [10]. Die TOSCA-Spezifikation [11] stellt eine Sprache zur Verfügung, die Service-Komponenten und ihre Beziehungen mithilfe einer Service-Topologie ("Service-Topology") beschreibt. Außerdem bietet sie noch die Beschreibung der Verwaltungsprozeduren an, welche die Services mittels Orchestrierungsprozesse ("Orchestration-Processes") erstellen, ändern und terminieren. In TOSCA werden diese Prozesse als Pläne bezeichnet. Die Kombination von Topologie und Orchestrierung in einem Service-Template beschreibt, was unter Deployments in verschiedenen Umgebungen benötigt wird. Das Ziel ist das interoperable Deployment von Cloud-Services und ihrer Verwaltung während des gesamten Lebenszyklus zu ermöglichen, wenn die Applikationen in unterschiedlichen Cloud-Umgebungen deployed werden.

2.2.1 TOSCA-Kernbegriffe

TOSCA ist eine XML-basierte Sprache und definiert ein Metamodell für das Spezifizieren von IT-Services. Dieses Metamodell legt die Struktur eines Service sowie die Art der Verwaltung fest. Ein Topolgy-Template (auch als Topologie-Modell eines Service bezeichnet) bestimmt die Struktur eines Service. Pläne [9] [15] definieren die Prozessmodelle, die verwendet werden, um einen Service zu erstellen, zu terminieren sowie ihn während seines ganzen Lebenszyklus zu verwalten.

Die wichtigsten Elemente, die einen Service definieren, sind in Abbildung 2.6 dargestellt.

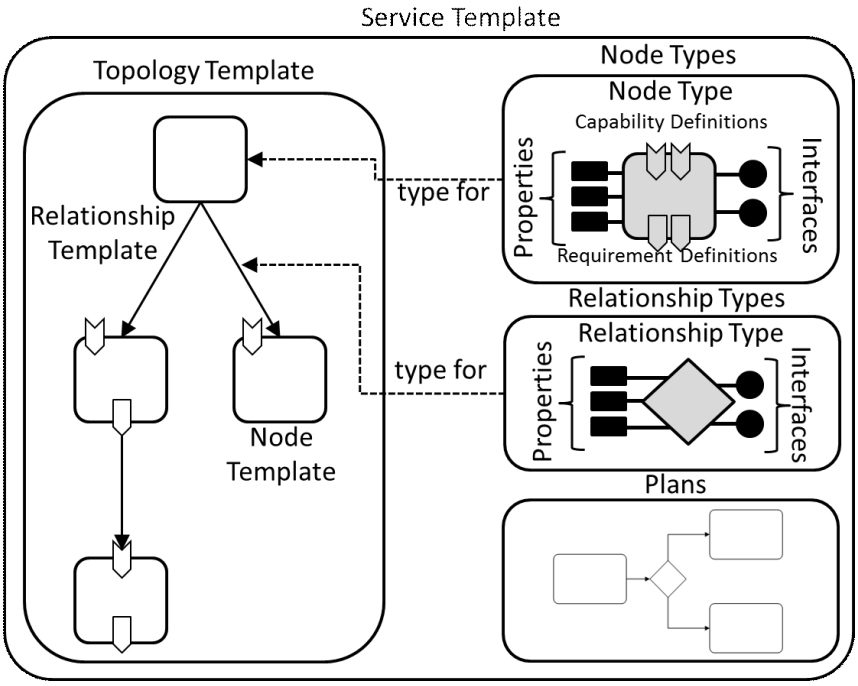

Abbildung 2.6: Strukturelle Elemente eines Service-Template und ihrer Beziehungen [8]

Ein Topology-Template besteht aus einer Reihe von Node-Templates und Relationship-Templates, die zusammen das Topologie-Modell eines Service als ein gerichteter Graph definieren. Ein Knoten in diesem Graph wird von einem Node-Template dargestellt. Ein Node-Template ist eine Instanz eines Node-Type. Ein Node-Type definiert die Eigenschaften einer solchen Komponente (via Node-Type-Properties) und die Operationen (via Interfaces), die das Deployment und Management eines Service ermöglichen. Node-Types sind für den Zweck der Wiederverwendung separat definiert.

Ein Relationship-Template spezifiziert die Beziehung zwischen zwei Knoten in einem Topology-Template. Jedes Relationship-Template bezieht sich auf einen Relationship-Type, der die Semantik und alle Eigenschaften der Beziehung definiert. Relationship-Types sind zum Wiederverwendungszweck separat definiert. Das Relationship-Template zeigt die verbundenen Elemente und die Richtung der Beziehung an, indem ein Quellelement und ein Zielelement (in geschachtelten "SourceElement" und "TargetElement" Elementen) definiert werden. Das Relationship-Template definiert auch alle möglichen Beschränkungen mit dem optionalen Element "RelationshipConstraints".

Pläne, die in einem Service-Template definiert sind, beschreiben die Verwaltungsaspekte von Service-Instanzen, insbesondere ihre Erstellung und Terminierung [23]. Diese Pläne sind als Prozessmodelle definiert, z.B. ein Workflow bestehend aus einem Schritt oder mehreren Schritten. Die Spezifikation ist abhängig von existierenden Sprachen wie BPMN [11] [15] oder BPEL [12], anstatt eine andere Sprache für das Definieren von Prozessmodellen anzubieten. Abhängigkeit von vorhandenen Standards in diesem Raum erleichtert Portabilität und Interoperabilität, aber alle Sprachen für das Definieren von Prozessmodellen können verwendet werden. Das TOSCA-Metamodell stellt Container zur Verfügung, entweder um ein Prozessmodell (via Plan-Model-Reference) zu referenzieren oder um ein Prozessmodell (via Plan-Model) einzubauen. Ein Prozessmodell kann die

Aufgaben (unter Verwendung von BPMN-Terminologie) beinhalten, die auf (1) die Operationen der Interfaces von Node-Templates (oder die Operationen, die von Node-Types definiert sind, und diese Node-Types sind im "type" Attribut der Node-Templates spezifiziert) oder (2) die Operationen der Interfaces von Relationship-Templates (oder die Operationen, die von Relationship-Types definiert sind, und diese Relationship-Types sind im "type" Attribut der Relationship-Templates spezifiziert) verweisen. Dabei kann ein Plan die Knoten der Topologie eines Service direkt manipulieren oder die Interaktion mit externen Systemen ausführen. Im Rahmen dieser Arbeit sind die Pläne nicht wichtig. Sie werden besprochen, um die strukturellen Elemente in einem Service-Template besser verstehen zu können. Diese Arbeit konzentriert sich auf das Topologie-Modell.

Um in einer bestimmten Umgebung die Durchführung und die Verwaltung des Lebenszyklus einer Cloud-Anwendung zu unterstützen, müssen alle entsprechenden Artefakte in dieser Umgebung verfügbar sein. Das heißt, dass neben dem Service-Template der Cloud-Anwendung die Deployment-Artefakte und die Implementation-Artefakte in dieser Umgebung verfügbar sein müssen [16]. Um die Verfügbarkeit von allen genannten Elementen zu garantieren, definiert diese Spezifikation ein entsprechendes Archiv-Format namens Cloud-Service-Archive (CSAR). Details über CSAR werden im nächsten Unterkapitel besprochen.

2.2.2 TOSCA Cloud Service Archive (CSAR)

Die Ausgabe der im Rahmen dieser Studienarbeit entwickelten Prozedur ist eine Datei, die als "CSAR" bezeichnet wird. Ein CSAR ist eine ZIP-Datei, die mindestens zwei Verzeichnisse enthält: "TOSCA-Metadata" und "Definitions". Darüber hinaus können andere Verzeichnisse in einer CSAR-Datei enthalten sein, d.h. der Ersteller einer CSAR-Datei hat die Freiheit, die Inhalte einer CSAR-Datei und die Strukturierung dieser Inhalte den Cloud-Anwendungen entsprechend zu definieren.

Das Verzeichnis "TOSCA-Metadata" enthält die Metadaten, welche die anderen Inhalte der CSAR-Datei beschreiben. Diese Metadaten werden als "TOSCA-Metadatei" bezeichnet. Diese Datei besitzt den Dateinamen "TOSCA.meta".

Das Verzeichnis "Definitions" enthält ein oder mehrere TOSCA-Definitions-Dokumente (Dateierweiterung **.tosca**). Diese "Definitions" Dateien enthalten in der Regel Definitionen bezüglich der Cloud-Anwendung der CSAR-Datei. Darüber hinaus kann eine CSAR-Datei nur die Definition der Elemente für Wiederverwendung in anderen Kontexten enthalten. Beispielsweise könnte eine CSAR-Datei verwendet werden, um eine Reihe von Node-Types und Relationship-Types mit ihren jeweiligen Implementierungen zu verpacken, die dann von Service-Templates in anderen CSAR-Dateien verwendet werden können. In den Fällen, wo eine komplette Cloud-Anwendung in einer CSAR-Datei verpackt ist, muss eins der Definitions-Dokumente im Verzeichnis "Definitions" eine Definition für Service-Template enthalten, die die Struktur und das Verhalten der Cloud-Anwendung definiert.

Abbildung 2.7 zeigt die Struktur einer CSAR-Datei. Die ersten zwei Verzeichnisse in der CSAR-Datei sind unabdingbar. Die übrigen kann der Ersteller dieser CASR-Datei frei gestalten.

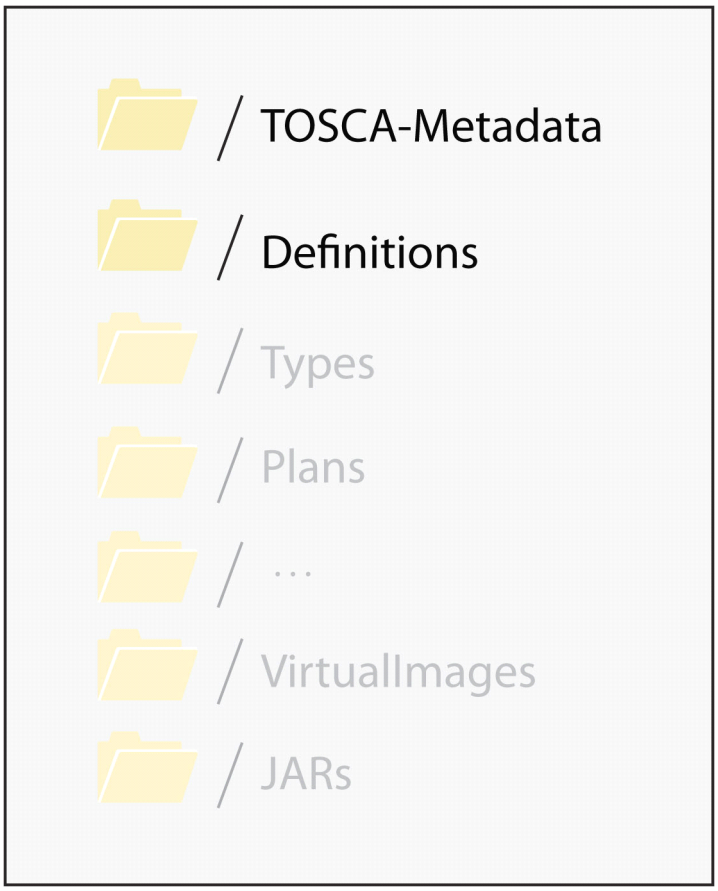

Abbildung 2.7: Die Struktur einer CSAR-Datei

2.2.3 TOSCA-Definitions Dokument

Alle Elemente, die zum Definieren eines TOSCA-Service-Template nötig sind, wie z.B. Node-Type-Definitionen, Relationship-Type-Definitionen sowie Service-Templates selbst, sind Teil eines TOSCA-Definitions-Dokuments. Dieser Abschnitt beschreibt die allgemeine Struktur eines TOSCA-Definitions-Dokuments. Ausschnitt 2.1 beschreibt ein Pseudo-Schema, das die XML-Syntax eines Definitions-Dokuments definiert. "?" bedeutet ein optionales Element oder Attribut. "*" bedeutet null oder mehrere Elemente bzw. Attribute. "+" bedeutet ein oder mehrere Element(e) bzw. Attribut(e). "|" bedeutet Auswählen. Zum Beispiel zeigt "a|b" eine Wahl zwischen "a" und "b". "(" und ")" werden verwendet, um den Rahmen der Operatoren "?", "*", "+" und "|" anzugeben.

```
01  <Definitions id="xs:ID"
02               name="xs:string"?
03               targetNamespace="xs:anyURI">
04
05      <Extensions>
06          <Extension namespace="xs:anyURI"
07                     mustUnderstand="yes|no"?/> +
08      </Extensions> ?
09
10      <Import namespace="xs:anyURI"?
11              location="xs:anyURI"?
12              importType="xs:anyURI"/> *
13
14      <Types>
15          <xs:schema .../> *
16      </Types> ?
17
18    (
19      <ServiceTemplate> ... </ServiceTemplate>
20    |
21      <NodeType> ... </NodeType>
22    |
23      <NodeTypeImplementation> ... </NodeTypeImplementation>
24    |
25      <RelationshipType> ... </RelationshipType>
26    |
27      <RelationshipTypeImplementation>...
            </RelationshipTypeImplementation>
28    |
29      <RequirementType> ... </RequirementType>
30    |
31      <CapabilityType> ... </CapabilityType>
32    |
33      <ArtifactType> ... </ArtifactType>
34    |
35      <ArtifactTemplate> ... </ArtifactTemplate>
36    |
37      <PolicyType> ... </PolicyType>
38    |
39      <PolicyTemplate> ... </PolicyTemplate>
40    ) +
41
42  </Definitions>
```

Ausschnitt 2.1: XML-Syntax eines TOSCA-Definitions-Dokuments [11]

Ein TOSCA-Definitions-Dokument muss mindestens eines der Elemente *ServiceTemplate*, *NodeType*, *NodeTypeImplementation*, *RelationshipType*, *RelationshipTypeImplementation*, *RequirementType*, *CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *PolicyType*, oder *PolicyTemplate*, definieren. Es kann aber beliebig viele dieser Elemente in einer beliebigen Reihenfolge definieren.

Diese Technik unterstützt eine modulare Definition von Service-Templates. Beispielsweise kann ein Definitions-Dokument nur die Definitionen von Node-Type und Relationship-Type enthalten, die dann in ein anderes Definitions-Dokument importiert werden können. Das zweite Definitions-Dokument definiert dann nur ein Service-Template und verwendet

die importierten Node-Types und Relationship-Types. Ebenso können Node-Type-Properties in separaten XML-Schema-Definitions-Dokumenten definiert werden, die bei dem Definieren eines Node-Type importiert und referenziert werden.

Im Folgenden werden die Elemente *Definitions*, *Import*, *RequirementType*, *CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *NodeType* und *NodeTypeImplementations* spezifiziert, die im Rahmen dieser Arbeit wichtig sind.

2.2.3.1 Definitions

Das Element *Definitions* ist das Wurzel-Element eines TOSCA-Definitions-Dokuments und hat die folgenden Attribute: *id*, *name* und *targetNamespace*. Das Attribut *id* spezifiziert den Bezeichner des Definitions-Dokuments, der innerhalb des Zielnamensraums ("Target Namespace") eindeutig sein muss. Das optionale Attribut *name* spezifiziert einen beschreibenden Namen des Dofinitions-Dokuments. Der Wert des Attributes *targetNamespace* spezifiziert den Zielnamensraum für das Definitions-Dokument. Alle Elemente, die innerhalb des Defnitions-Dokuments definiert sind, werden zu diesem Zielnamensraum hinzugefügt, außer ein Element besitzt ein eigenes targetNamespace-Attribut. Dann gilt der darin angegebene Namensraum.

2.2.3.2 Import

Das Element *Import* deklariert eine Abhängigkeit von externen TOSCA-Definitionen, XML-Schema-Definitionen oder WSDL-Definitionen [14]. Eine beliebige Anzahl von Elementen *Import* kann als Kinder des Elements *Definitions* erscheinen. Das Element *Import* hat die folgenden Attribute: *namespace*, *location* und *importType*. Das optionale Attribut *namespace* spezifiziert eine absolute URI, die die importierten Definitions-Dokumente identifiziert. Das optionale Attribut *location* enthält eine URI, die angibt, wo sich das relevante Definitions-Dokument befindet. Das erforderliche Attribut *importType* identifiziert den Typ des Dokuments, das durch eine absolute URI importiert wird. Ein Definitions-Dokument muss alle verwendeten Node-Types, Node-Type-Implementations, Relationship-Types, Relationship-Type-Implementations, Requirement-Types, Capability-Types, Artifact-Types, Policy-Types, WSDL-Definitionen und XML-Schema-Definitionen definieren oder importieren.

2.2.3.3 Requirement Types

Ein Requirement-Type ist eine wiederverwendbare Entität, die eine Art Anforderung ("Requirement") beschreibt. Ein Node-Type kann deklarieren, solche Anforderung zu besitzen. Zum Beispiel kann ein Requirement-Type für eine Datenbankverbindung definiert werden. Verschiedene Node-Types (z.B. ein Node Type für eine Anwendung) können deklarieren, eine Anforderung für eine Datenbankverbindung zu besitzen.

Das Element *RequirementType* hat die folgenden wichtigen Attribute: *name*, *targetNamespace* und *requiredCapabilityType*. Das Attribut *name* spezifiziert den Namen oder den Bezeichner des Requirement-Type, der innerhalb des Zielnamensraums eindeutig sein muss. Das optionale Attribut *targetNamespace* spezifiziert den Zielnamensraum, zu dem die Definition des Requirement-Type hinzugefügt werden wird. Wenn *targetNamespace* nicht angegeben wird, wird die Definition des Requirement-Type zum Zielnamensraum des Definitions-Dokuments, in dem dieser Requirement-Type definiert ist,

hinzugefügt werden. Das optionale Attribut *requiredCapabilityType* spezifiziert den Typ einer Fähigkeit ("Capability") also die Erfüllung einer Anforderung, der dem definierten Requirement-Type entsprechen muss. Der Wert dieses Attributs verweist auf den Namen eines *CapabilityType*-Elements, das in demselben Definitions-Dokument oder in einem separaten importierten Dokument definiert wird.

2.2.3.4 Capability Types

Ein Capability-Type ist eine wiederverwendbare Entität, die eine Art Fähigkeit ("Capability") beschreibt. Ein Node-Type kann deklarieren, solche Fähigkeit bereitzustellen. Zum Beispiel kann ein Capability-Type für einen Datenbankserver definiert werden. Verschiedene Node-Types (z.B. ein Node-Type für eine Datenbank) können deklarieren, die Fähigkeit eines Datenbankservers zur Verfügung zu stellen.

Das Element *CapabilityType* hat die folgenden wichtigen Attribute: *name* und *targetNamespace*. Das Attribut *name* spezifiziert den Namen oder den Bezeichner des Capability-Type, der innerhalb des Zielnamensraums eindeutig sein muss. Das optionale Attribut *targetNamespace* spezifiziert den Zielnamensraum, zu dem die Definition des Capability-Type hinzugefügt werden wird. Wenn *targetNamespace* nicht angegeben wird, wird die Definition des Capability-Type zum Zielnamensraum des Definitions-Dokuments, in dem dieser Capability-Type definiert ist, hinzugefügt werden.

2.2.3.5 Artifact Types

Ein Artifact-Type ist eine wiederverwendbare Entität, die die Art eines Artifact-Template oder von meheren Artifact-Templates definiert. Diese Artifact-Templates dienen als Deployment-Artefakte für Node-Templates oder als Implementation-Artefakte für die Interface-Operationen von Node-Type und Relationship-Type. Zum Beispiel könnte ein Artifact-Type "WAR-Datei" zur Beschreibung von Web-Application-Archive-Files definiert werden. Auf der Grundlage des Artifact-Type können ein oder mehrere Artifact-Templates, die konkrete WAR-Dateien darstellen, definiert und als Deployment- oder Implementation-Artefakte referenziert werden. Ein Artifact-Type kann die Struktur von beobachtbaren Eigenschaften durch eine Properties-Definition definieren, d.h. die Namen , die Datentypen und die erlaubten Werte, die die Eigenschaften, die in Artifact-Templates definiert sind, haben können. Diese Artifact-Templates benutzen einen Artifact-Type oder Instanzen von solchen Artifact-Templates.

Das Element *ArtifactType* hat die folgenden wichtigen Eigenschaften: das Attribut *name*, das Attribut *targetNamespace* und das Element *PropertiesDefinition*. Das Attribut *name* spezifiziert den Namen oder den Bezeichner des Artifact-Type, der innerhalb des Zielnamensraum eindeutig sein muss. Das optionale Attribut *targetNamespace* spezifiziert den Zielnamensraum, zu dem die Definition des Artifact-Type hinzugefügt werden wird. Wenn *targetNamespace* nicht angegeben wird, wird die Definition des Artifact-Type zum Zielnamensraum des Definitions-Dokuments, in dem dieser Artifact-Type definiert ist, hinzugefügt werden. Das Element *PropertiesDefinition* spezifiziert mittels XML-Schema die Struktur der beobachtbaren Eigenschaften des Artifact-Type, wie seine Konfiguration und sein Zusand. Dieses Element hat nur eines der beiden Attribute *element* und *type*. Das Attribut *element* gibt den Namen eines XML-Elements an, das die Struktur der Artifact-

Type-Properties definiert. Das Attribut *type* gibt den Namen eines (komplexen) XML-Typs an, der die Struktur der Artifact-Type-Properties definiert.

2.2.3.6 Artifact Templates

Ein Artifact-Template beschreibt ein Artefakt, das von anderen Objekten in einem Service-Template als ein Deployment- oder Implementation-Artefakt referenziert werden kann. Von Node-Types oder Node-Templates könnte beispielsweise ein Artifact-Template für einige installierbaren Software als ein Deployment-Artefakt für das Instanziieren einer spezifischen Software-Komponente referenziert werden. Als ein weiteres Beispiel könnte aus den Definitionen für das Interface der Node-Types oder der Relationship-Types ein Artifact-Template für eine WAR-Datei als Implementation-Artefakt für eine REST-Operation referenziert werden.

Ein Artifact-Template bezieht sich auf einen spezifischen Artifact-Type, der die Struktur von beobachtbaren Eigenschaften (Metadaten) oder das Artefakt definiert. Das Artifact-Template definiert in der Regel die Werte dieser Eigenschaften innerhalb des Elements *Properties*. Außerdem stellt in der Regel ein Artifact-Template eine Referenz oder mehrere Referenzen auf das tatsächliche Artefakt selbst zur Verfügung. Es kann als eine Datei in der CSAR-Datei sein, welche das gesamte Service-Template enthält. Es kann auch an einem entfernten Ort wie einem FTP-Server verfügbar sein.

Das Element *ArtifactTemplate* hat die folgenden wichtigen Attribute: *id*, *name* und *type*. Das Attribut *id* spezifiziert den Bezeichner des Artifact-Template, der innerhalb des Zielnamensraum eindeutig sein muss. Das optionale Attribut *name* spezifiziert den Namen des Artifact-Template. Der Wert des Attributs *type* verweist auf einen Artifact-Type, der den Typ des Artifact-Template zur Verfügung stellt.

Das Element *ArtifactTemplate* hat die folgenden wichtigen Kindelemente: *Properties* und *ArtifactReferences*. Das optionale Element *Properties* spezifiziert die invarianten Eigenschaften des Artifact-Template, d.h. die Eigenschaften, die allgemein in verschiedenen Kontexten verwendet werden, in denen das Artifact-Template benutzt wird. Das optionale Element *ArtifactReferences* enthält einen Verweis oder mehrere Verweise auf die tatsächlichen Artefakte. Jeder Verweis wird durch ein separates Element *ArtifactReference* dargestellt. Das Element *ArtifactReference* hat wichtige Eigenschaften wie *reference* und *Include*. Das Attribut *reference* enthält eine URI, die auf ein tatsächliches Artefakt zeigt. Wenn diese URI eine relative URI ist, wird sie relativ zum Wurzelverzeichnis der CSAR-Datei, die das Service-Template enthält, interpretiert. Das optionale Element *Include* kann verwendet werden, um ein Pattern der Dateien zu definieren. Diese Dateien sind in dem gesamten Artefakt-Verweis ("Artifact Reference") enthalten, falls *reference* auf ein komplettes Verzeichnis verweist. Das Element *Include* hat ein Attribut *pattern*. Dieses Attribut enthält eine Pattern-Definition für die Dateien, die in dem gesamten Artefakt-Verweis eingeschlossen sind.

2.2.3.7 Node Types

Ein Node-Type ist eine wiederverwendbare Entität, die die Art eines Node-Template oder von mehreren Node-Templates definiert. Ein Node-Type definiert die Struktur der beobachtbaren Eigenschaften durch eine Properties-Definition, d.h. die Namen, die

Datentypen und die zulässigen Werte, die die Eigenschaften, die in Node-Templates definiert sind, haben können. Diese Node-Templates benutzen einen Node-Type oder die Instanzen von solchen Node-Templates. Im Folgenden werden die Eigenschaften des Elements *NodeType* spezifiziert, die im Rahmen dieser Arbeit wichtig sind.

Das Attribut *name* spezifiziert den Namen oder den Bezeichner des Node-Type, der innerhalb des Zielnamensraums eindeutig sein muss. Das optionale Attribut *targetNamespace* spezifiziert den Zielnamensraum, zu dem die Definition des Node-Type hinzugefügt werden wird. Wenn *targetNamespace* nicht angegeben wird, wird die Definition des Node-Type zum Zielnamensraum des Definitions-Dokuments, in dem dieser Node-Type definiert ist, hinzugefügt werden.

Durch das Element *PropertiesDefinition* kann die Struktur der beobachtbaren Eigenschaften des Node-Type, wie seine Konfiguration und sein Zustand, mittels XML-Schema spezifiziert. Dieses Element besitzt genau eines der beiden Attribute *element* und *type*. Das Attribut *element* gibt den Namen eines XML-Elements an, das die Struktur der Node-Type-Properties definiert. Das Attribut *type* gibt den Namen eines (komplexen) XML-Typ an, der die Struktur der Node-Type-Properties definiert. In Ausschnitt 2.2 wird ein Beispiel für ein Node-Type-Properties-Dokument dargestellt.

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://jujucharms.com/charms/precise/myblg/..."
  targetNamespace="http://jujucharms.com/charms/precise/myblog/...">
  <xs:complexType name="t-myblog-properties">
    <xs:sequence>
      <xs:element name="port" type="int" default="80">
        <xs:annotation>
          <xs:documentation xml:lang="en">
            Port to listen on
          </xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="admin-email" type="string" default="null">
        <xs:annotation>
          <xs:documentation xml:lang="en">
            Email address for the site administrator.
          </xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="myblog-properties" type="t-myblog-properties"/>
</xs:schema>
```

Ausschnitt 2.2: XML-Schema für ein Node-Type-Properties-Dokument

Ein Node-Type kann deklarieren, bestimmte Anforderungen ("Requirements") mittels des Elements *RequirementDefinition* zu benötigen. Das Element *RequirementDefinition* hat wichtige Attribute wie *name* und *requirementType*. Das Attribut *name* spezifiziert den Namen der definierten Anforderung und muss innerhalb von *RequirementsDefinitions* des

aktuellen Node-Type eindeutig sein. Das Attribut *requirementType* identifiziert durch seinen Wert den Requirement-Type, welche durch das aktuelle Element *RequirementDefinition* definiert wird.

Außerdem kann ein Node-Type deklarieren, bestimmte Fähigkeiten ("Capabilities") unter Verwendung von dem Element *CapabilityDefinition* bereitzustellen. Das Element *CapabilityDefinition* hat die wichtigen Attribute: *name* und *capabilityType*. Das Attribut *name* spezifiziert den Namen der definierten Fähigkeit und muss innerhalb von *CapabilityDefinition* des aktuellen Node-Type eindeutig sein. Das Attribut *capabilityType* identifiziert durch seinen Wert den Capability-Type, welche durch das aktuelle Element *CapabilityDefinition* definiert wird.

Die Funktionen, die auf (einer Instanz von) einem entsprechenden Node-Template durchgeführt werden können, werden durch die Interfaces des Node-Type definiert. Das Element *Interfaces* enthält die Definitionen der Operationen, die auf (Instanzen von) dem Node-Type durchgeführt werden können. Solche Definitionen der Operationen werden in Form von verschachtelten Elementen *Interface* angegeben. Das Element *Interface* hat ein Attribut *name* und enthält ein Element *Operation*. Das Attribut *name* beschreibt den Namen des Interface. Der Name ist eine URI, die im Rahmen des definierenden Node-Type eindeutig sein muss. Das Element *Operation* definiert eine verfügbare Operation, um besondere Aspekte des Node-Type wie z.B. den Lebenszyklus eines Service zu verwalten. Dieses Element hat ein Attribut *name*. Dieses Attribut definiert den Namen der Operation und muss innerhalb des Interface, das die Operation enthält, eindeutig sein.

2.2.3.8 Node Type Implementations

Eine Node-Type-Implementation beschreibt den ausführbare Code, der einen spezifischen Node-Type implementiert. Die Node-Type-Implementation stellt auch eine Sammlung von ausführbaren Dateien oder Programmen zur Verfügung, welche die Interface-Operationen eines Node-Type (auch bekannt als Implementation-Artefakte) implementieren. Außerdem stellt er eine Sammlung von ausführbaren Dateien oder Programmen zur Verfügung, die nötig sind, um die Instanzen von Node-Templates, die sich auf einen bestimmten Node-Type (auch bekannt als Deployment-Artefakte) beziehen, zu erstellen. Diese ausführbaren Dateien oder Programme werden als separate Artifact-Templates definiert und von den Implementation-Artefakte und den Deployment-Artefakte eines Node-Type referenziert.

Das Element *NodeTypeImplementation* hat die folgenden wichtigen Attribute: *name*, *nodeType* und *targetNamespace*. Das Attribut *name* spezifiziert den Namen oder den Bezeichner der Node-Type-Implementation, der innerhalb des Zielnamensraums eindeutig sein muss. Das optionale Attribut *targetNamespace* spezifiziert den Zielnamensraum, zu dem die Definition der Node-Type-Implementation hinzugefügt werden wird. Wenn *targetNamespace* nicht angegeben wird, wird die Node-Type-Implementation zum Zielnamensraum des Definitions-Dokuments, in dem diese Node-Type-Implementation definiert ist, hinzugefügt werden. Der Wert des Attributs *nodeType* spezifiziert den Node-Type, der durch diese Node-Type-Implementation implementiert wird.

Das Element *ImplementationArtifacts* spezifiziert eine Reihe von Implementation-Artefakten für Interfaces oder Operationen eines Node-Type. Jedes Implementation-

Artefakt eines Interface oder einer Operation wird durch das Kindelement *ImplementationArtifact* spezifiziert. Dieses Kindelement hat die folgenden Attribute: *name*, *artifactType*, *artifactRef*, *interfaceName* und *operationName*. Das Attribut *name* spezifiziert den Namen des Artefakts, der im Rahmen dieser Node-Type-Implementation eindeutig sein soll. Das Attribut *artifactType* spezifiziert den Typ des Artefakts. Sein Wert soll dem Namen eines in demselben Definitions-Dokument oder in einem importierten Dokument definierten Elements *ArtifactType* entsprechen. Das optionale Attribut *artifactRef* enthält einen Namen, der ein Artifact-Template als ein Implementation-Artefakt identifiziert. Dieses Artifact-Template kann in demselben Definitions-Dokument oder in einem separaten, importierten Dokument definiert werden. Das optionale Attribut *interfaceName* spezifiziert den Namen des Interface, das durch das tatsächlichen Implementation-Artefakt implementiert wird. Das optionale Attribut *operationName* spezifiziert den Namen der Operation, die durch das tatsächliche Implementation-Artefakt implementiert wird.

Die Laufzeitumgebungen, die TOSCA unterstützen, werden als TOSCA-Containers bezeichnet. Ein TOSCA-Container muss eine Reihe von den Typen der Implementation-Artefakte verarbeiten, die verwendet werden um die Verwaltungsoperationen (zum Beispiel das Instanziieren eines Node-Type) auszuführen. Außerdem soll ein TOSCA-Container auch eine Reihe von den Typen der Deployment-Artefakte, die der TOSCA-Container verarbeiten kann, unterstützen, weil es für das Instanziieren eines Node-Type erforderlich ist, die Deployment-Artefakte in der entsprechenden Umgebung zur Verfügung zu stellen. Die Node-Type-Implementations können durch das Element *RequiredContainerFeatures* die Hinweise für einen TOSCA-Container spezifizieren, dass er eine Implementierung, die einer bestimmten Umgebung entspricht, richtig auswählen kann.

3 Entwurf

In Kapitel 2 wurden die Details über die Eingabe "Juju-Charm", die Ausgabe "TOSCA-CSAR" der Prozedur sowie die Grundlagen von Juju und TOSCA besprochen. Dadurch ist die Grundlage geschaffen, sich mit dem Entwurf der Prozedur zu beschäftigen. Zuerst wird die zu implementierende Prozedur analysiert, um genauer zu erläutern, wie der Zusammenhang zwischen der Eingabe und der Ausgabe der Prozedur aussieht. Als nächster Schritt wird das Konzept der Prozedur dargestellt, die beschreibt, welche Funktionen für die Prozedur implementiert werden müssen und in welcher Reihenfolge die Schritte der Prozedur ablaufen sollen. Außerdem werden der Entwuf der zu implementierenden Pakete in der Prozedur und die Software von Drittanbietern beschrieben. Bevor wir auf die Implementierung der Prozedur eingehen, wird auch ein Sequenzdiagramm zum besseren Verstehen für die Implementierung besprochen.

3.1 Analyse der Prozedur

In diesem Abschnitt wird der Zusammenhang zwischen der Eingabe und der Ausgabe der Prozedur besprochen. In Abbildung 3.1 werden die Informationen bezüglich dieser Arbeit dargestellt. "xxx" ist der Name eines Service.

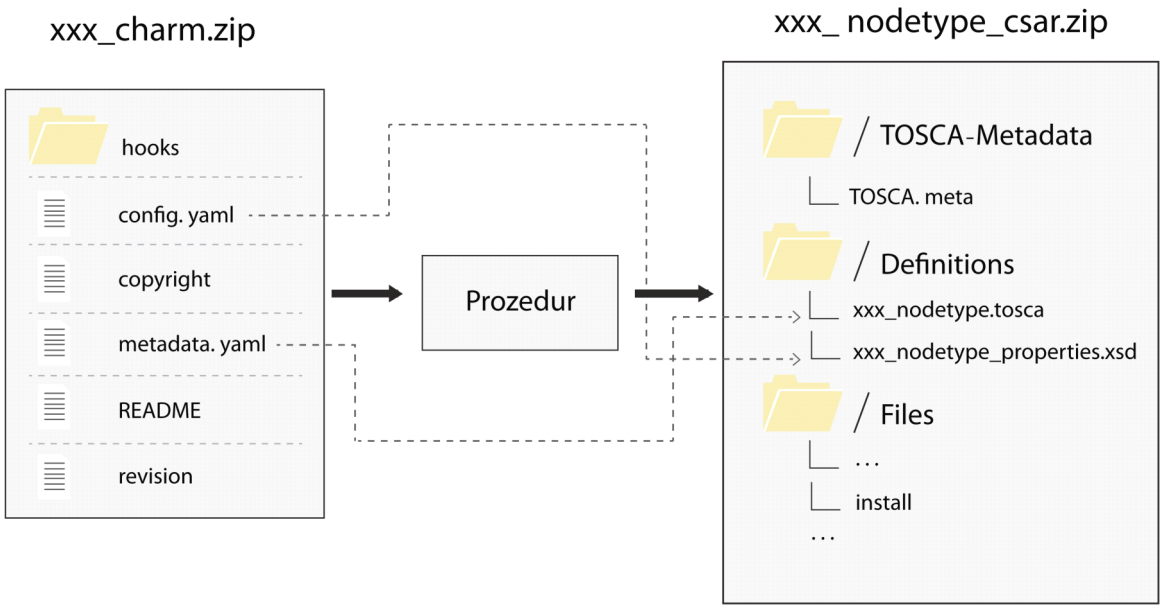

Abbildung 3.1: Zusammenhang zwischen der Eingabe und der Ausgabe der Prozedur

In dieser Abbildung kann man sehen, wie die Struktur sowie die Inhalte der Eingabe und der Ausgabe der Prozedur aussehen. Die Datei "metadata.yaml" ist für unsere Arbeit sehr wichtig. Durch die Inhalte dieser Datei, zum Beispiel die Informationen über "name", "requires" und "provides", kann unsere Prozedur ein entsprechendes TOSCA-Definitions-Dokument für einen Node-Type generieren. Ebenso kann die Prozedur durch die Informationen der Datei "config.yaml" auch ein entsprechendes Node-Type-Properties-Dokument erzeugen. In der Abbildung gibt es zwei gestrichelte Linien, die zeigen, dass die Prozedur die Dateien "metadata.yaml" und "config.yaml" in der Datei "xxx_charm.zip"

einliest und dann die entsprechenden Dateien "xxx_nodetype.tosca" und "xxx_nodetype_properties.xsd" generiert. Darüber hinaus müssen alle originale Dateien (alle Hooks und die anderen Dateien) in der Datei "xxx_charm.zip" zu der von unserer Prozedur generierten Datei "xxx_nodetype_csar.zip" kopiert werden. Dort werden sie noch als die entsprechenden Artefakte (die ausführbaren Codes oder Dateien) verwendet werden, um in der TOSCA-Umgebung den Lebenszyklus einer Cloud-Anwendung durchzuführen und zu verwalten [23]. Wo sich diese Dateien in der CSAR-Datei befinden sollen, kann der Ersteller der CSAR-Datei selbst entscheiden. Im Rahmen dieser Arbeit werden sie alle in dem Verzeichnis "Files" gespeichert. Schließlich muss noch eine wichtige Metadatei "TOSCA.meta" erstellt und sie zur CSAR-Datei hinzugefügt werden.

3.2 Konzept der Prozedur

Mindestens drei Funktionen muss die Prozedur implementieren, um die Aufgabe dieser Arbeit zu erledigen. Erstens muss die Prozedur die ZIP-Datei bearbeiten. Beispielsweise kann sie eine Datei in eine ZIP-Datei einbauen oder eine Datei aus einer ZIP-Datei ausnehmen. Zweitens muss die Prozedur auch die YAML-Dateien behandeln. Sie kann die Informationen aus einer YAML-Datei bekommen und diese Informationen auf einer gewissen Weise speichern, damit diese Informationen später noch benutzt werden kann. Schließlich muss die Prozedur die XML-Datei erstellen. In unserem Fall soll die Prozedur nur das XML-basierte TOSCA-Definitions-Dokument mittels der Informationen aus der YAML-Datei generieren.

Für das Schreiben der Prozedur ist noch wichtig in welcher Reihenfolgen die Schritte der Prozedur ablaufen sollen. Dazu werden im Allgemeinen die folgenden Schritte benötigt:

Schritt 1: Die Charm-ZIP-Datei einlesen und die entsprechenden YAML-Dateien erhalten.

Schritt 2: Die YAML-Dateien analysieren und ihre Inhalte auf einer gewissen Weise speichern.

Schritt 3: Die XML-Dateien durch die Inhalte der YAML-Dateien erzeugen.

Schritt 4: Schließlich die CSAR-Datei generieren.

In Abbildung 3.2 wird ein einfaches Pseudo-Sequenzdiagramm für die Prozedur dargestellt. Diese Abbildung zeigt, dass die Funktionseinheit "ZIP-File-Handler" zuerst von der Funktionseinheit "CSAR-Generator" aufgerufen wird. "ZIP-File-Handler" liest eine "xxx_charm.zip" Datei ein und gibt die Dateien "metadata.yaml" und "config.yaml" zurück. Durch die zwei YAML-Dateien erzeugt die Funktionseinheit "Juju-Yaml-Reader" zwei abstrakte Objekte. Jedes Objekt wird hier als "YamlModelList" bezeichnet. Diese Objekte dienen zum Speichern der Inhalte der YAML-Dateien. Die Funktionseinheit "XML-Generator" kann durch die zwei YamlModelList-Objekte die entsprechenden, XML-basierten Dateien "xxx_nodetype.tosca" und "xxx_nodetype_properties.xsd" generieren. Schließlich wird die Funktionseinheit "ZIP-File-Handler" wieder aufgerufen, um die Zieldatei "xxx_nodetype_csar.zip" zu erzeugen. Diese Zieldatei enthält nicht nur die zwei generierten XML-Dateien sondern auch alle Dateien in der "xxx_charm.zip" Datei und noch eine entsprechende Metadatei "TOSCA.meta".

Wenn es in mancher Charms keine "config.yaml" Datei gibt, dann wird die entsprechende Bearbeitung für diese Datei ignoriert.

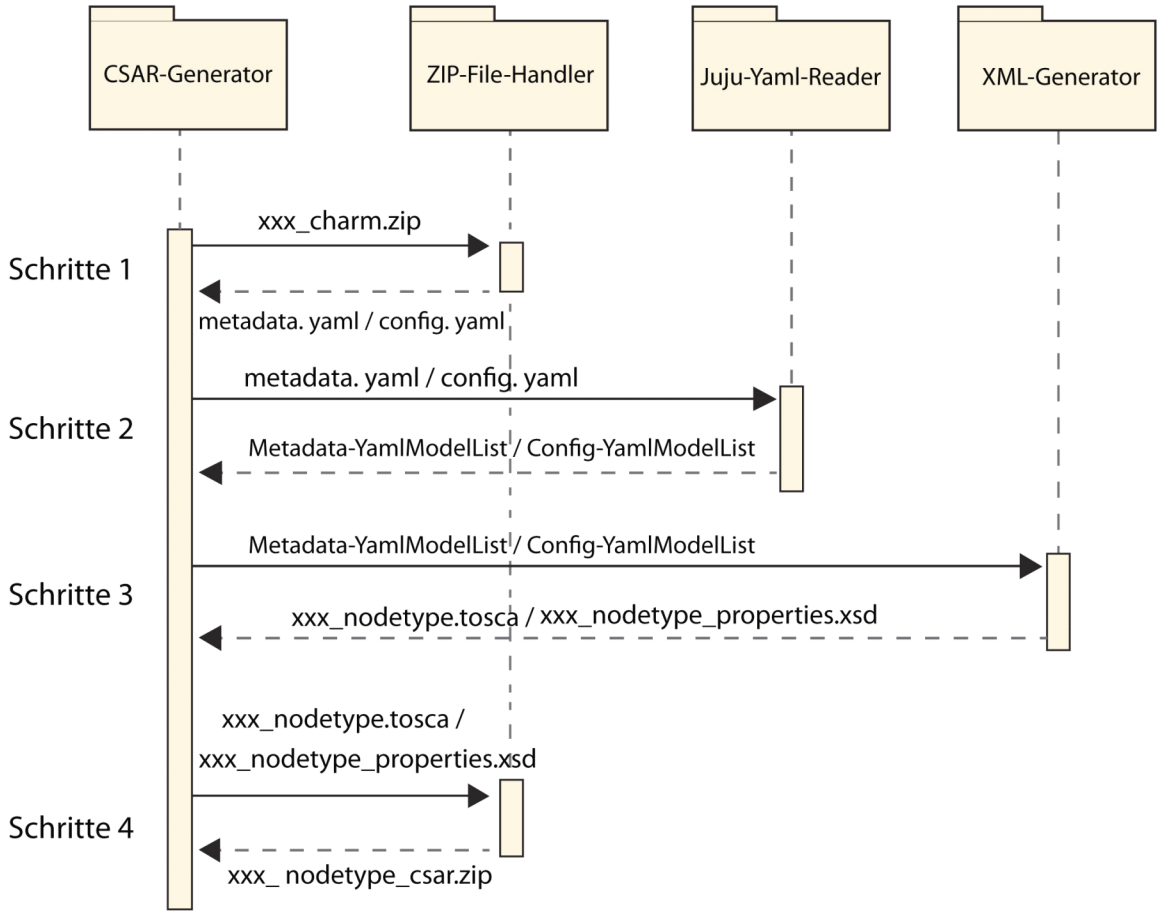

Abbildung 3.2: Ein einfaches Pseudo-Sequenzdiagramm der Prozedur

3.3 Entwurf der zu implementierenden Pakete

In Unterkapitel 3.2 werden vier Funktionseinheiten erwähnt. Dazu werden die folgenden entsprechenden Pakete entworfen: "org.tosca", "org.tosca.xml", "org.tosca.yaml" und "org.tosca.zip". Jedes Paket wird als eine Funktionseinheit angesehen und kann die bestimmte Funktion der Prozedur implementieren.

3.3.1 TOSCA-CSAR-Generator

Die Funktion des Pakets "org.tosca" ist, die anderen Pakete der bestimmten Reihenfolge nach aufzurufen, um eine TOSCA-CSAR-Datei zu generieren. Außerdem müssen hier der Pfadname und der Dateiname der Eingabe und der Ausgabe der Prozedur angegeben werden.

3.3.2 ZIP-File-Handler

Die Funktion des Pakets "org.tosca.zip" ist die Verarbeitung einer ZIP-Datei. Mögliche Aufgaben sind zum Beispiel, eine Liste, die eine Reihe von Pfadnamen und Dateinamen aller Dateien in einer ZIP-Datei enthält, zu erhalten, eine Datei zu einer ZIP-Datei hinzuzufügen oder eine Datei aus einer ZIP-Datei auszunehmen. Außerdem kann damit die

Struktur der Dateiverzeichnisse in einer ZIP-Datei nach der bestimmten Form geändert werden.

3.3.3 Juju-Yaml-Reader

Die Funktion des Pakets "org.tosca.yaml" ist, eine YAML-Datei zu lesen. Beispielsweise können die Inhalte einer YAML-Datei analysiert und jede wichtige Information aus dieser YAML-Datei als ein YamlModel-Objekt in einer Liste gespeichert werden.

3.3.4 TOSCA-XML-Generator

Die Funktion des Pakets "org.tosca.xml" ist, eine XML-Datei zu erstellen. In unserem Fall werden ein TOSCA-Definitions-Dokument und ein XML-Schema-Definitions-Dokument erzeugt. Das Paket ermöglicht, dass durch die Liste mit YamlModel-Objekten alle entsprechenden Elemente für die zwei oben genannten Dokumente im Rahmen dieser Arbeit gespeichert werden und dadurch die entsprechenden XML-Dokumente generiert werden.

3.4 Verwendetes Java-Paket und Software von Drittanbietern

3.4.1 JDK

Für die Implementierung von Schritt 1 und 4 wird das Java-Paket "java.util.zip" [17] verwendet. Dieses Paket stellt die Klassen für das Lesen und das Schreiben von ZIP-Dateien zur Verfügung. Man benutzt das Paket, um die Dateien in der ZIP-Datei zu lesen und eine neue ZIP-Datei zu generieren. Für unseren Fall kann die Prozedur die Dateien "metadata.yaml" und "config.yaml" in einer Charm-ZIP-Datei erhalten und sie an einem bestimmten Ort speichern. Schließlich kann die Prozedur durch das Paket eine neue CSAR-ZIP-Datei erzeugen und dabei ein neu generiertes TOSCA-Definitions-Dokument zu diesem CSAR-ZIP-Datei hinzufügen.

3.4.2 SnakeYaml

Für Schritt 2 wird eine YAML-Software von Drittanbietern verwendet. SnakeYAML [18] ist ein YAML-Parser für Programmiersprache "Java". SnakeYAML ist bekannt dafür, dass er ein kompletter YAML1.1-Parser ist. Durch SnakeYAML kann die YAML-Datei gelesen werden und können ihre Inhalte gespeichert werden.

3.4.3 Dom4j

Für Schritt 3 wird eine XML-Software von Drittanbietern benutzt. Dom4j [19] ist eine benutzerfreundliche, open-source-Bibliothek, die dafür verwendet wird, mit XML auf der Java-Plattform unter Verwendung von Java-Collections-Framework zu arbeiten. Um genauer zu sagen, ist Dom4j eine Java-XML-API, die zum Lesen und Schreiben von XML-Dateien verwendet wird. Unter Verwendung von DOM4j und Informationen in der Yaml-Datei kann man eine auf TOSCA-Standard basierende XML-Datei erzeugen.

3.5 Komplettes Sequenzdiagramm

Bevor man sich mit der Implementierung der Prozedur beschäftigt, wird hier zuerst ein komplettes Seuquenzdiagramm in Abbildung 3.3 dargestellt, in dem der gesamte Ablauf der Prozedur und die Interaktion ihrer Komponenten gezeigt werden [20]. In der

Abbildung sind die wichtigen Klassen und die wichtigen Methoden zu sehen, die implementiert werden müssen. Für Übersichtlichkeit werden die Parameter der Methoden weggelassen. Dem Sequenzdiagramm nach kann die Prozedur schrittweise implementiert werden.

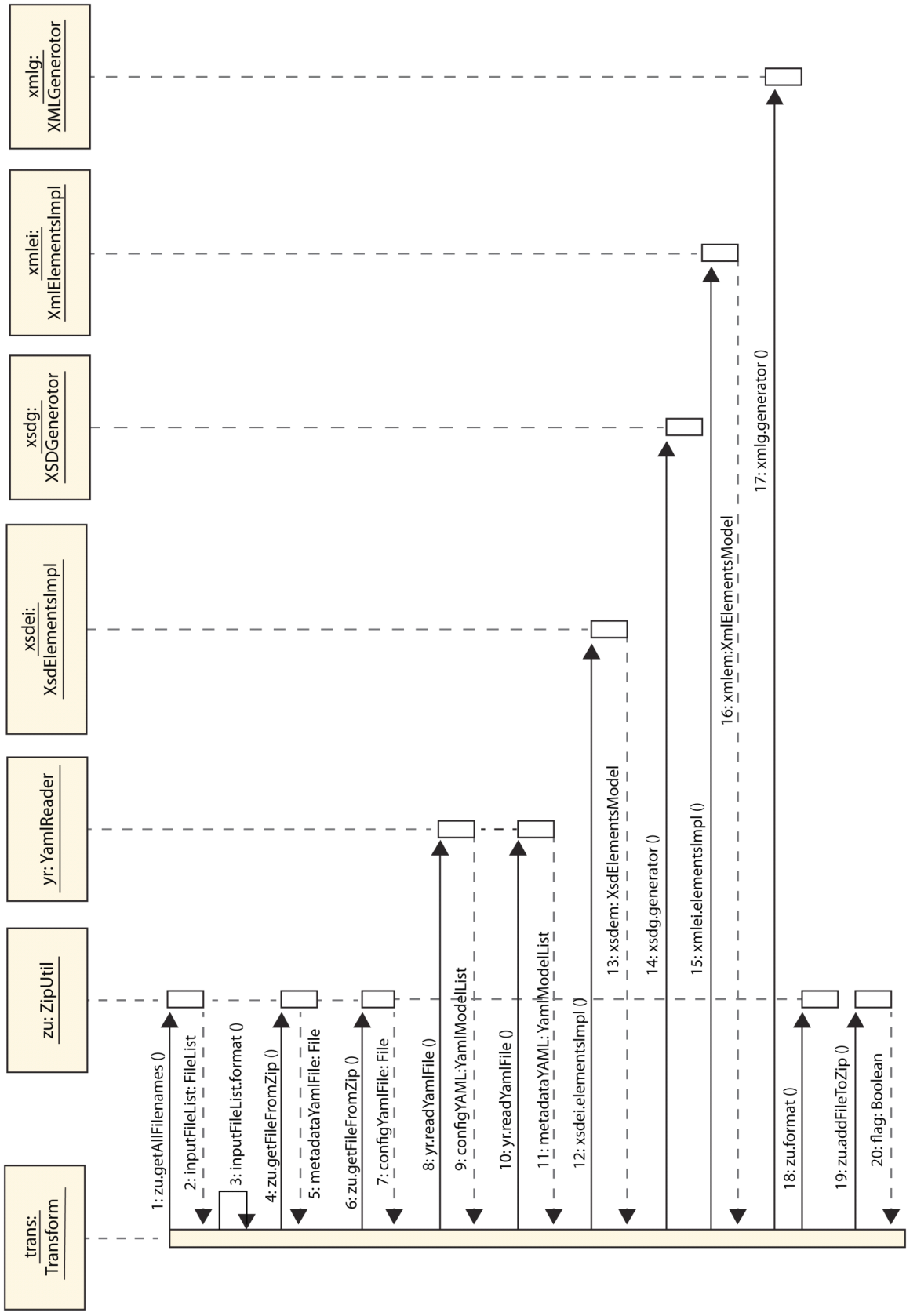

Abbildung 3.3: Ein Sequenzdiagramm der Prozedur

4 Implementierung

In diesem Kapitel wird auf Details der Implementierung der Prozedur eingegangen. In den folgenden Abschnitten wird die Implementierung von allen wichtigen Klassen und Methoden ausführlich beschrieben.

4.1 Implementirung des Pakets "org.tosca"

Das Paket "org.tosca" enthält im Rahmen dieser Arbeit nur eine Klasse "CharmToNodeType".

4.1.1 Klasse "CharmToNodeType"

In dieser Klasse werden die Methode "main" und eine Klasse "Transform" definiert.

4.1.1.1 Methode "main"

Die Methode "main" [21] ist die Einstiegsfunktion der Prozedur und wird automatisch als erste Funktion aufgerufen. In dieser Methode definiert man drei Eigenschaften vom Typ "String". Dies sind "pathname", "inputFilename" und "platform". Die letzte Eigenschaft "platform" beschreibt eine bestimmte Umgebung, von der eine Implementierung eines Node-Type abhängig ist. Die Information über "platform" wird in Kapitel 2.2.3.8 besprochen. Die anderen zwei Eigenschaften stellen die Pfadenamen und die Dateinamen der Eingabe der Prozedur dar. Außerdem wird in der Methode "main" eine Instanz der Klasse "Transform" mit den drei definierten Eigenschaften als Parameter erstellt und ihre Methode "transform" aufgerufen.

4.1.1.2 Klasse "Transform"

Diese Klasse enthält einen Konstruktor [21] "Transform" und eine Methode "transform". Der Konstruktor "Transform" wird mit drei Parametern versehen, die von der Methode "main" übergeben werden. Durch diese Parameter werden allen nötigen Variablen die Werte zugewiesen. Die Funktion der Methode "transform" ist, die Instanzen von allen notwendigen Klassen zu erstellen und ihre Methoden der bestimmten Reihenfolge nach aufzurufen, um die Aufgabe dieser Prozedur zu verwirklichen. Die Reihenfolge, in der die Methoden aufgerufen werden, wird auf der linken Seite in Abbildung 3.3 dargestellt.

4.2 Implementierung des Pakets "org.tosca.zip"

In dem Paket "org.tosca.zip" werden drei Klassen implementiert: "FileModel", "FileModelList" und "ZipUtil".

4.2.1 Klasse "FileModel"

Die Funktion dieser Klasse ist, den Pfadnamen und den Dateinamen einer Datei als ein FileModel-Objekt zu speichern. In dieser Klasse werden zwei privaten Eigenschaften "pathname" und "filename" definiert, deren Werte durch Setter- und Getter-Methoden jeweils zugewiesen und erhalten werden können.

4.2.2 Klasse "FileModelList"

Die Funktion dieser Klasse ist, die Pfadnamen und die Dateinamen aller Dateien in einer ZIP-Datei als FileModel-Objekte in einem FileModelList-Objekt zu speichern. In dieser Klasse wird eine private Eigenschaft "fileModelList" vom Typ "java.util.List" [22] definiert. In der Konstrukturmethode wird die Eigenschaft "fileModelList" instanziiert, das zur Speicherung der FileModel-Objekte dient. Die Werte der Eigenschaft "fileModelList" können durch Setter- und Getter-Methoden zugewiesen und bekommen werden. Außerdem enthält diese Klasse noch eine Methode "format",

4.2.2.1 Methode "format"

Input/Output	**Parametername**	**Parametertyp**	**Beschreibung**
Input	"servicename"	String	Der Name eines Service
Input	"platform"	String	Eine Umgebung, von der eine Implementierung von "NodeType" abhängen kann
Output	"fl"	FileList	Eine Liste von den Pfadnamen und Dateinamen aller Dateien in einer ZIP-Datei

Tabelle 4.1: Parameter der Methode "format"

In dieser Methode wird der Pfadname jeder Datei in der Charm-Datei zu dem in der CSAR-Datei verlangten Pfadnamen geändert. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.1 gezeigt. Die zwei Parameter der Methode werden in den neuen Pfadenamen benötigt. Wenn in der Charm-Datei die Dateien namens "install", "start" und "stop" vorhanden sind, müssen zusätzlich die entsprechenden Dateien names "install.sh", "start.sh"und "stop.sh" zur CSAR-Datei hinzugefügt werden. Der Grund dafür ist, dass nicht bekannt ist, in welcher Script-Sprache die Dateien "install", "start" und "stop" implementiert wurden. Bei TOSCA muss man definieren, um welche Art von Script (zum Beispiel Shell oder Python) es sich handelt [11]. Deswegen braucht man die zusätzliche Datei "install.sh" als Wrapper-Script, um die Datei "install" aufzurufen. Dasselbe gilt auch für "start" und "stop" [23]. Außerdem müssen diese Dateien in Form von FileModel-Objekten mit den neuen Pfadnemen und den originalen Dateinamen zu dem entsprechenden FileList-Objekt "fl" hinzugefügt werden. Am Ende gibt die Methode das FileList-Objekt "fl" zurück, das alle FileModel-Objekte der CSAR-Datei entsprechend enthält.

4.2.3 Klasse "ZipUtil"

Die Funktion dieses Pakets ist die Verarbeitung einer ZIP-Datei. Zu den Aufgaben gehören zum Beispiel eine Dateiliste, die eine Reihe von Pfadnamen und Dateinamen aller Dateien in einer ZIP-Datei enthält, zu bekommen, eine Datei zu einer ZIP-Datei hinzuzufügen oder eine Datei aus einer ZIP-Datei zu nehmen. Außerdem kann durch diese

Klasse die Struktur der Dateiverzeichnisse in einer ZIP-Datei der bestimmten Form nach geändert werden.

4.2.3.1 Methode "getAllFilenames"

Input/Output	Parametername	Parametertyp	Beschreibung
Input	"inputFilename"	String	Der vollständige Name einer ZIP-Datei
Output	"fl"	FileList	Eine Liste der Pfadnamen und Dateinamen aller Dateien in einer ZIP-Datei

Tabelle 4.2: Parameter der Methode "getAllFilenames"

Die Funktion dieser Methode ist, eine Dateiliste mit den Pfadnamen und den Dateinamen aller Dateien in einer ZIP-Datei zurückzugeben. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.2 gezeigt. Zuerst wird eine Instanz der Klasse "ZipInputStream" [24] erstellt, die einen Input-Stream für das Lesen der Dateien in der ZIP-Datei implementiert. Mit der Methode "getNextEntry" der Klasse "ZipInputStream" kann der nächste ZIP-Datei-Eintrag gelesen und dann ein Objekt der Klasse "ZipEntry" als Ausgabe zurückgegeben werden. Durch eine While-Schleife kann man dann alle Dateien in der ZIP-Datei in Form der ZipEntry-Objekte erhalten. Bei jeder Schleife wird die Methode "getName" der Klasse "ZipEntry" [24] aufgerufen, damit man einen vollständigen Namen jeder Datei bekommen kann. Durch das Teilen jedes vollständigen Namen kann man den Pfadnamen und den Dateinamen erhalten, die als ein FileModel-Objekt in einem FileList-Objekt "fl" gespeichert werden. Schließlich wird das FileList-Objekt "fl" zurückgegeben.

4.2.3.2 Methode "getFileFromZip"

Input/Output	Parametername	Parametertyp	Beschreibung
Input	"inputFile"	String	Der vollständige Name einer ZIP-Datei, aus der eine Datei genommen wird
Input	"filename"	String	Der Name einer Datei, die aus einer ZIP-Datei genommen wird
Output	"output"	File	Eine Datei, die aus der ZIP-Datei genommen wird

Tabelle 4.3: Parameter der Methode "getFileFromZip"

Die Funktion dieser Methode ist, eine Datei mit einem angegebenen Dateinamen aus einer ZIP-Datei zu erhalten. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.3 gezeigt. Zuerst erstellt man zwei Instanzen der Klasse "ZipInputStream" und "FileOutputStream" [25]. Die Klasse "ZipInputStream" implementiert einen Input-Stream für das Lesen der Dateien in einer ZIP-Datei. Die Klasse "FileOutputStream" [24] implementiert einen Output-Stream für das Schreiben der Daten in eine Datei. Ähnlich wie die Implementierung der oben besprochenen Methode "getAllFilenames" kann man durch das Teilen eines vollständigen Namens den Pfadnamen und den Dateinamen einer Datei erhalten. Falls dieser Dateiname dem erwarteten Dateinamen entspricht, wird die Datei mit diesem Dateinamen durch ein Objekt der Klasse "FileOutputStream" in die Datei "output" geschrieben. Schließlich wird die Datei "output" zurückgegeben.

4.2.3.3 Methode "format"

Input/Output	**Parametername**	**Parametertyp**	**Beschreibung**
Input	"input"	String	Pfad zu der zu ändernden ZIP-Datei
Input	"output"	String	Pfad zu der geänderten ZIP-Datei
Input	"newPathname"	String	Für das Definition eines neuen Pfadnamen verwendet

Tabelle 4.4: Parameter der Methode "format"

Die Funktion dieser Methode ist, die Struktur der Dateiverzeichnisse einer ZIP-Datei zu ändern und dann eine neue ZIP-Datei mit der neuen Struktur der Dateiverzeichnisse zu erzeugen. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.4 gezeigt. Tatsächlich wird nur die Pfadnamen jeder Datei in der ZIP-Datei geändert. Zuerst erstellt man zwei Instanzen der Klasse "ZipInputStream" und "ZipOutputStream". Die Klasse "ZipInputStream" implementiert einen Input-Stream für das Lesen der Dateien in einer ZIP-Datei. Die Klasse "ZipOutputStream" implementiert einen Output-Stream für das Schreiben der Dateien in eine ZIP-Datei. Ähnlich wie die Implementierung der oben besprochenen Methode "getAllFilenames" kann man die Pfadnamen und die Dateinamen aller Dateien erhalten. Dann kann man die Pfadnamen ändern. Schließlich müssen alle Dateien als die neu instanziierten Objekte der Klasse "ZipEntry" mit den neuen Pfadnamen und den originalen Dateinamen durch das ZipOutputStream-Objekt in die neue ZIP-Datei geschrieben werden.

4.2.3.4 Methode "addFileToZip"

Input/Output	Parametername	Parametertyp	Beschreibung
Input	"xmlFileArg"	String	Der Name der XML-Datei, der zur ZIP-Datei hinzugefügt wird
Input	"xsdFileArg"	String	Der Name der XSD-Datei, der zur ZIP-Datei hinzugefügt wird
Input	"zipInFileArg"	String	Der Name einer ZIP-Datei, zu der die neuen Dateien hinzugefügt werden
Input	"zipOutFileArg"	String	Der Name einer ZIP-Datei, zu der die neuen Dateien schon hinzugefügt wurden
Output	"flag"	Boolean	Ob das Hinzufügen der Dateien erfolgreich ist.

Tabelle 4.5: Parameter der Methode "addFileToZip"

Die Funktion der Methode ist, Dateien zu einer ZIP-Datei hinzuzufügen. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.5 gezeigt. Wenn in der Charm-ZIP-Datei keine "config.yaml" Datei vorhanden ist, d.h. es wird kein entsprechendes Node-Type-Properties-Dokument generiert, dann wird die Methode ohne Parameter "xsdFileArg" benutzt. In dem anderen Fall, dass es in der Charm-ZIP-Datei die beiden Dateien "metadata.yaml" und "config.yaml" gibt, d.h. es werden das Node-Type-Properties-Dokument und das TOSCA-Definitions-Dekument für Node-Type erzeugt, dann muss die Methode mit den Parameter "xsdFileArg" und "xmlFileArg" verwendet werden. Am Anfang wird jede Datei in einer Eingabe-ZIP-Datei als ein Zip-Input-Stream gelesen und dann als ein Zip-Output-Stream in eine neue ZIP-Datei geschrieben. Danach werden die XML-Datei und die XSD-Datei zu der neuen ZIP-Datei hinzugefügt. Außerdem muss noch eine Datei namens "TOSCA.meta" erzeugt und zu dieser neu generierten ZIP-Datei hinzugefügt. Schließlich wird "true" zurückgegeben, wenn keine Ausnahmen passiert haben.

4.3 Implementierung des Pakets "org.tosca.yaml"

In dem Paket "org.tosca.yaml" werden drei Klassen implementiert: "YamlModel", "YamlModelList" und "YamlReader".

4.3.1 Klasse "YamlModel"

Die Funktion dieser Klasse ist, die Informationen in der YAML-Datei zu speichern. In einer YAML-Datei erscheinen alle Informationen in Form von Schlüssel-Wert-Paaren. Jede Information kann als ein YamlModel-Objekt nämlich ein Schlüssel-Wert-Paar

gespeichert werden. In dieser Klasse werden drei privaten Eigenschaften "key", "value" und "yamlValue" definiert, deren Werte durch Setter- und Getter-Methoden zugewiesen und erhalten werden können. Die Eigenschaften "key" und "value" sind vom Typ "String" Die Eigenschaft "yamlValue" ist vom Typ "YamlModel" und kann noch ein YamlModel-Objekt enthalten. In Abbildung 4.1 wird gezeigt, wie jede Information in der "metadata.yaml" Datei des Service "mysql" als ein YamlModel-Objekt gespeichert wird.

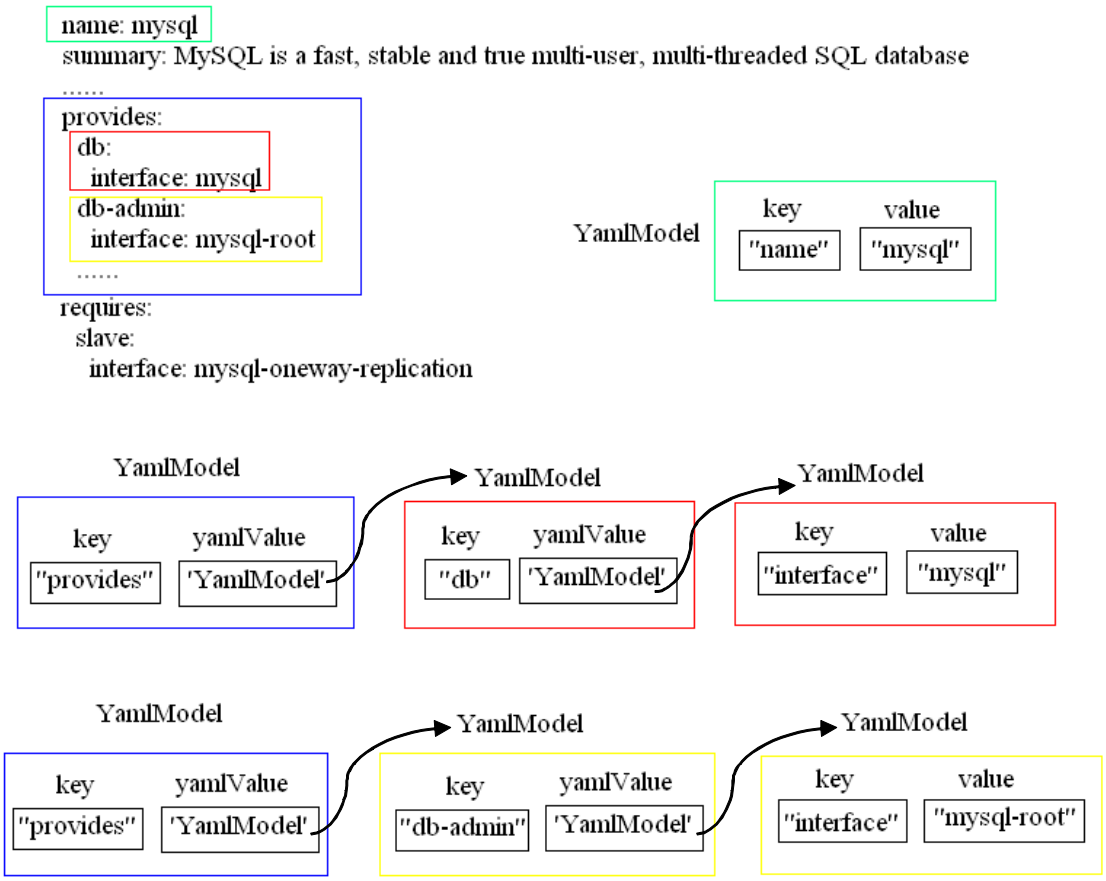

Abbildung 4.1: Das Speichern von Informationen in einer "metadata.yaml" Datei

4.3.2 Klasse "YamlModelList"

Die Funktion dieser Klasse ist, alle Informationen in einer YAML-Datei als YamlModel-Objekte in einem YamlModelList-Objekt zu speichern. In dieser Klasse wird eine private Eigenschaft "yamlModelList" vom Typ "java.util.List" definiert. In dem Konstruktor wird die Eigenschaft "yamlModelList" instanziiert, die für das Speichern der YamlModel-Objekte verwendet wird. Die Werte der Eigenschaft "yamlModelList" können durch Setter- und Getter-Methoden zugewiesen und erhalten werden.

4.3.3 Klasse "YamlReader"

Die Funktion dieser Klasse ist, die Inhalte einer YAML-Datei zu lesen. Dafür wird eine Software "SnakeYaml" eines Drittanbieters verwendet.

4.3.3.1 Methode "readYamlFile"

Durch "SnakeYaml" kann diese Methode die Inhalte in einer YAML-Datei in Form vom Typ "java.util.Map" [22] lesen. Die Methode benötigt einen Eingabeparameter "file" vom Typ "java.io.File" [25]. Der Parameter bedeutet eine YAML-Datei, die bearbeitet wird. Als Ausgabe liefert diese Methode ein YamlModelList-Objekt. Um die Ausgabe zu erzeugen, werden zuerst die Instanzen der Klasse "Yaml" und "FileInputStream" erstellt. Durch eine Methode "load" [27] der Klasse "Yaml" kann ein Java-Objekt vom Typ "java.util.Map" zurückgegeben werden. Nach dem Aufruf einer privaten Methode "read" mit dem Eingabeparater vom Typ "java.util.Map" wird ein YamlModelList-Objekt erzeugt.

4.3.3.2 Methode "read"

Durch diese Methode können die Inhalte in einem Objekt vom Typ "java.util.Map" gelesen und dann als YamlModel-Objekte gespeichert werden. Alle YamlModel-Objekte werden in einem YamlModelList-Objekt gelagert, welche schließlich zurückgegeben werden muss. Zuerst bekommt die Methode als Eingabeparameter ein Objekt vom Typ "java.util.Map". Dann wird der Typ des Objekts zum Typ "java.util.Set" [22] verwandelt, um die Elemente von "java.util.Map" direkt durch einen Iterator [22] ausgeben zu können. Jedes Element in "java.util.Map" ist in Form von einem "key-value" Paar gespeichert. Für den Typ von "value" gibt es zwei Möglichkeiten. Wenn "value" der Typ "java.util.LinkedHashMap" [22] ist, d.h. "value" enthält auch eine Reihe der Objekte vom Typ "YamlModel", dann wird die Methode "read" mit dem Eingabeparameter "value" rekursiv aufgerufen. Das Ziel ist YamlModel-Objekte in "value" zu bekommen. Diese Objekte mit dem gleichen "key" zusammen werden als YamlModel-Objekte zum YamlModelList-Objekt hinzugefügt. In dem anderen Fall, dass "value" den elementaren Datentyp wie "String", "Integer" oder "Boolean" hat, dann werden "key" und "value" als ein YamlModel-Objekt zum YamlModelList-Objekt hinzugefügt.

4.4 Implementierung des Pakets "org.tosca.xml"

In dem Paket "org.tosca.xml" werden folgende Klassen implementiert: "XmlElementsModel", "XmlElementsImpl", "XmlGenerator", "XsdElementsModel", "XsdElementsImpl" und "XsdGenerator". Die ersten drei Klassen dienen dazu, ein XML-Dokument zu generieren. Die anderen drei Klassen werden verwendet, um ein XSD-Dokument zu erzeugen. Da XSD auf XML basiert und zur Beschreibung der Struktur eines XML-Dokuments dient, haben diese Klassen die ähnliche Implementierung.

4.4.1 Klasse "XmlElementsModel"

Die Funktion dieser Klasse ist, die Elemente in einer XML-Datei zu speichern. Es handelt sich um alle Elemente sowie ihre Kinderelemente in einem TOSCA-Definitions-Dokument, die schon in Unterkapitel 2.2.3 besprochen wurden. Zum Beispiel werden die Elemente *RequirementType*, *CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *NodeType* und *NodeTypeImplementation* als private Eigenschaften in dieser Klasse definiert und können durch Setter- und Getter-Methoden zugewiesen und erhalten werden.

4.4.2 Klasse "XmlElementsImpl"

Die Funktion dieser Klasse ist im Allgemeinen, durch die Inhalte in einer "metadata.yaml" Datei die entsprechenden Elemente für ein TOSCA-Definitions-Dokument zu generieren.

Das heißt, durch das Lesen eines YamlModelList-Objekts ein XmlElementsModel-Objekt zu erzeugen. Am Anfang wird ein privates Objekt der Klasse "XmlElementsModel" in dem Konstruktor instanziiert, das zum Speichern von Elementen dient. Durch das Aufrufen der Methode "elementsImpl" werden alle Elemente in dem TOSCA-Defnitions-Dokument implementiert. Jedes Element wird durch eine entsprechende Methode implementiert. Beispielsweise wird das Root-Element *Definitions* durch die Methode "rootImpl" implementiert und das Element *NodeType* durch die Methode "nodeTypeImpl". Für ihre Implementierung werden zwei Klassen "org.dom4j.DocumentHelper" und "org.dom4j.Namespace" sowie ein Interface "org.dom4j.Element" von Dom4j [26] verwendet. Ein Element kann durch die Methode "createElement" der Klasse "DocumentHelper" generieren.

4.4.2.1 Methode "elementsImpl"

Input/Output	**Parametername**	**Parametertyp**	**Beschreibung**
Input	"yml"	YamlModelList	Ein YamlModellist-Objekt
Input	"fl"	FileModelList	Ein FileModelList-Objekt
Input	"flag"	Boolean	Bedeutet, ob in der Charm-Zip-Datei eine "config.yaml" Datei vorhanden ist.
Input	"platform"	String	Eine Umgebung, von der eine Implementierung von "NodeType" abhängen kann
Output	"xmlem"	XmlElements-Model	Ein XmlElementsModel-Objekt

Tabelle 4.6: Parameter der Methode "elementsImpl"

Die Funktion dieser Methode ist, durch ein YamlModelList-Objekt die YamlModel-Objekte zu bekommen und dadurch die entsprechenden Elemente zu generieren. Ihre Eingabeparameter und ihre Ausgabe werden in Tabelle 4.6 gezeigt. In der Regel benötigt man drei wichtige Informationen in einer "metadata.yaml" Datei. Diese drei Informationen beziehen sich auf drei YamlModel-Objekte mit den Schlüsselwerten "name", "requires" und "provides". Alle generierten Elemente werden in einem Objekt der Klasse "XmlElementsModel" gespeichert. Schließlich wird dieses XmlElementsModel-Objekt zurückgegeben.

Durch das YamlModel-Objekt, dessen Schlüssel den Wert "name" hat, kann man den Namen des Cloud-Service bekommen. Dabei können die Methoden "rootImpl", "importImpl", "artifactTypeImpl" und "nodeTypeImpl" aufgerufen werden, um die entsprechenden Elemente *Definitions* (Root-Element), *Import*, *ArtifactType* und *NodeType* zu erzeugen. Bevor man die Methoden "artifactTypeImpl" und "nodeTypeImpl" aufruft,

müssen noch die Namen der Operationen für den Lebenszyklus eines Cloud-Service, wie zum Beispiel "install", "start" und "stop", aus einem FileModelList-Objekt bekommen werden. Diese Operationsnamen werden danach als Parameter vom Typ "java.util.List" auf die Methode "artifactTypeImpl" und "nodeTypeImpl" übertragen.

Durch die YamlModel-Objekte mit dem Schlüsselwert "requires" kann man die Informationen über das Element *RequirementType* erhalten. Damit können das Element *RequirementType* sowie seine Attribute durch das Aufrufen einer entsprechenden Methode "requirementTypeImpl" erzeugt werden. Das Element *RequirementType* hat drei Attribute: *name*, *targetNamespace* und *requiredCapabilityType*. Für das Definieren des Attributs *requiredCapabilityType* muss man zuerst überprüfen, ob es in der Liste, die die Elemente *CapabilityType* enthält, ein entsprechendes Element gibt, das denselben Namen mit dem Element *RequirementType* hat. Wenn die Liste des Elements *CapabilityType* leer ist oder es kein solches Element *CapabilityType* gibt, dann braucht man das Attribut nicht zu definieren.

In ähnlicher Weise können durch das YamlModel-Objekt "provides" die Informationen über das Element *CapabilityType* bekommen und damit dieses Element und seine Attribute generiert werden. Hier wird eine Methode "capabilityTypeImpl" aufgerufen. Das Element *CapabilityType* hat zwei Attribute: *name* und *targetNamespace*. Man muss noch überprüfen, ob ein Element in der Liste des Elements *RequirementType* vorhanden ist, das denselben Namen mit diesem Element *CapabilityType* hat. Wenn es ein solches Element *RequirementType* gibt, dann wird ein entsprechendes Attribut *requiredCapabilityType* mit dem Namen des Elements *CapabilityType* zum Element *RequirementType* hinzugefügt.

4.4.2.2 Methode "rootImpl"

In dieser Methode wird das Root-Element *Definitions* erzeugt. Dabei werden seine Attribute *name*, *targetNamespace* und *id* zu diesem Element hinzugefügt. Dann definiert man auch einige Namensräume, die für dieses Dokument benötigt sind. Die Methode benötigt zwei Parameter: "name" und "flag". Der Parameter "name" bedeutet den Namen eines Service. Der Parameter "flag" hat den Typ "Boolean" und bedeutet, ob in der Charm-ZIP-Datei eine "config.yaml" Datei vorhanden ist. Wenn "flag" wahr ist, bedeutet das, dass es ein Node-Type-Properties-Dokument in der TOSCA-CSAR-Datei gibt, das durch die Datei "config.yaml" erzeugt wird. Dabei muss ein entsprechender Namensraum definiert werden. Schließlich wird das Root-Element in dem XmlElementsModel-Objekt gespeichert.

4.4.2.3 Methode "importImpl"

In dieser Methode wird das Element *Import* erzeugt. Die Methode benötigt zwei Parameter "name" und "flag". Wenn "flag" wahr ist, muss ein Node-Type-Properties-Dokument importiert werden. Deshalb muss ein neues Element *Import* mit dem Attribut *location* erzeugt werden. Das Attribut *location* beschreibt den Pfad zu dem importierten Dokument. Schließlich wird ein *Import*-Element oder mehrere *Import*-Elemente in dem XmlElementsModel-Objekt gespeichert.

4.4.2.4 Methode "requirementTypeImpl"

In dieser Methode werden das Element *RequirementType* und seine Attribute erzeugt. Die Parameter dieser Methode dienen zum Definieren der Attribute, wenn sie nicht NULL sind. Nachdem das Element *RequirementType* in dem XmlElementsModel-Objekt gespeichert ist, wird eine Methode "requirementDefinitionsImpl" aufgerufen, die zur Implementierung der Kinderelemente *RequirementDefinitions* des Elements *NodeType* dient.

4.4.2.5 Methode "capabilityTypeImpl"

In dieser Methode wird das Element *CapabilityType* und seine Attribute erzeugt. Die Implementierung dieser Methode ist gleich wie die Methode "requirementTypeImpl". Darüber wird hier nicht redundant gesprochen. Schließlich wird eine Methode "capabilityDefinitionsImpl" aufgerufen, die zur Implementierung der Kinderelemente *CapabilityDefinitions* des Elements *NodeType* dient.

4.4.2.6 Methode "artifactTypeImpl"

In dieser Methode wird das Element *ArtifactType* und seine Attribute erzeugt. Nach dem Erzeugen des Elements und seiner Attribute wird die Methode "artifactTypePropertiesDefinitionImpl" aufgerufen, um sein Kindelement *PropertiesDefinition* zu implementieren. Dann wird dieses Element in dem Objekt der Klasse XmlElementsModel gespeichert. Die Methode "artifactTypeImpl" benötigt zwei Eingabeparameter: "fl" und "operationNames". Der Parameter "operationNames" vom Typ "java.util.List" bedeutet die Namen der Operationen für den Lebenszyklus eines Cloud-Service. Für jede Operation wie "install", "start" oder "stop" muss eine Methode "artifactTemplateImpl" aufgerufen werden, um das entsprechende Element *ArtifactTemplate* zu implementieren. Der Parameter "fl" wird als ein Parameter an der Methode "artifactTemplateImpl" weitergegeben.

4.4.2.7 Methode "artifactTemplateImpl"

In dieser Methode wird das Element *ArtifactTemplate* und seine Attribute erzeugt. Für das Attribut *id* wird die Klasse "java.util.UUID" [22] verwendet. Die Funktion der Klasse "java.util.UUID" ist, einen unveränderlichen "Universally-Unique-Identifier" (UUID) zu generieren. Ein UUID beschreibt einen 128-Bit-Wert. Dann wird dieses Element in dem XmlElementsModel-Objekt gespeichert. Darüber hinaus muss man die Methode "artifactTemplatePropertiesImpl" zur Generierung seines Kindelements *Properties* und die Methode "artifactReferencesImpl" zur Erzeugung seines anderen Kindelements *ArtifactReferences* aufrufen. Schließlich wird zur Implementierung der Kindelemente *ImplementationArtifact* des Elements *NodeTypeImplementation* noch eine Methode "implementationArtifactImpl" aufgerufen.

4.4.2.8 Methode "nodeTypeImpl"

In dieser Methode werden das Element *NodeType* und seine Attribute generiert. Einer der Parameter der Methode besitzt den Namen "flag" und er ist vom Typ "Boolean". Wenn "flag" wahr ist, gibt es in der TOSCA-CSAR-Datei ein Node-Type-Properties-Dokument. Dazu muss die Methode "nodeTypePropertiesDefinitionImpl" aufgerufen werden, um sein Kindelement *PropertiesDefinition* zu erzeugen. Außerdem muss zum Generieren seines Kindelements *Interfaces* die Methode "interfacesImpl" aufgerufen werden. Schließlich

wird die Methode "nodeTypeImplementationImpl" aufgerufen, um das Element *NodeTypeImplementation* zu erzeugen.

4.4.2.9 Methode "nodeTypeImplementationImpl"

In dieser Methode werden das Element *NodeTypeImplementation* und seine Attribute generiert. Einer der Parameter dieser Methode besitzt den Namen "platform". Wenn der Parameter "platform" nicht NULL ist, bedeutet das, dass eine bestimmte Umgebung, von der das Element *NodeTypeImplementation* abhängt, angegeben ist. In diesem Fall wird die Methode "requiredContainerFeaturesImpl" aufgerufen, um das Kindelement *requiredContainerFeatures* zu erstellen. Schließlich wird zum Generieren des Kindelements *ImplementationArtifacts* die Methode "implementationArtifactsImpl" aufgerufen.

4.4.3 Klasse "XmlGenerator"

Die Funktion dieser Klasse ist, durch ein XmlElementsModel-Objekt das endgültige XML-Dokument, nämlich das TOSCA-Definitions-Dokument zu erzeugen. Das TOSCA-Definitions-Dokument enthält alle durch die Methode "elementsImpl" der Klasse "XmlElementsImpl" generierten Elemente. Zuerst wird in dem Konstruktor ein XmlElementsModel-Objekt empfangen, in dem alle Elemente gespeichert ist. Außerdem wird auch eine Methode "generator" definiert, um ein XML-Dokument zu generieren.

4.4.3.1 Methode "generator"

Die Funktion dieser Methode ist, durch die Methode "createDocument" der Klasse "org.dom4j.DocumentHelper" ein XML-Dokument zu erstellen. Dabei müssen das Root-Element und seine Kindelemente zu diesem Dokument hinzugefügt werden. Die Methode "generator" benötigt einen Eingabeparameter "output", der zeigt, wo das generierte XML-Dokument ausgegeben werden sollte.

4.4.4 Klasse "XsdElementsModel"

Die Funktion dieser Klasse ist, die Elemente "root", "xs:complexType", "xs:sequence" und "xs:element" in einem XSD-Datei zu speichern. Diese Elemente werden als private Eigenschaften definiert und können durch Setter- und Getter-Methoden zugewiesen und erhalten werden.

4.4.5 Klasse "XsdElementsImpl"

Die Funktion dieser Klasse ist im Allgemeinen, durch die Inhalte in einer "config.yaml" Datei die entsprechenden Elemente für ein Node-Type-Properties-Dokument (ein XSD-Dokument) zu generieren. Das heißt, durch das Lesen eines YamlModelList-Objekts ein XsdElementsModel-Objekt zu erzeugen. Am Anfang wird ein privates Objekt der Klasse "XsdElementsModel" in dem Konstruktor instanziiert, das zum Speichern von Elementen dient. Außerdem werden noch eine Klasse "ConfigModel" zum Speichern der Optionen in einer "config.yaml" Datei definiert. Durch das Aufrufen der Methode "elementsImpl" werden alle Elemente in dem Node-Type-Properties-Dokument implementiert. Für die Elementen "root", "xs:complexType", "xs:sequence" und "xs:element" werden die entsprechenden Methoden "rootImpl", "complexTypeImpl", "sequenceImpl" und "elementImpl" definiert.

4.4.5.1 Klasse "ConfigModel"

Die Funktion der Klasse ist, die Optionen in einer "config.yaml" Datei zu speichern. Jede Option mit einem eigenen Optionsnamen besteht aus drei Teilen: "default", "type" und "description". Jeder Teil bezieht sich auf ein YamlModel-Objekt. Durch das Lesen eines YamlModelList-Objekts für eine "config.yaml" Datei wird jede Option als ein ConfigModel-Objekt gespeichert. In Abbildung 4.2 wird gezeigt, wie jede Option in der "config.yaml" Datei als ein ConfigModel-Objekt gespeichert wird. Dazu definiert man in der Klasse "ConfigModel" vier private Eigenschaften vom Typ "String": "optionname", "default", "type" und "description".

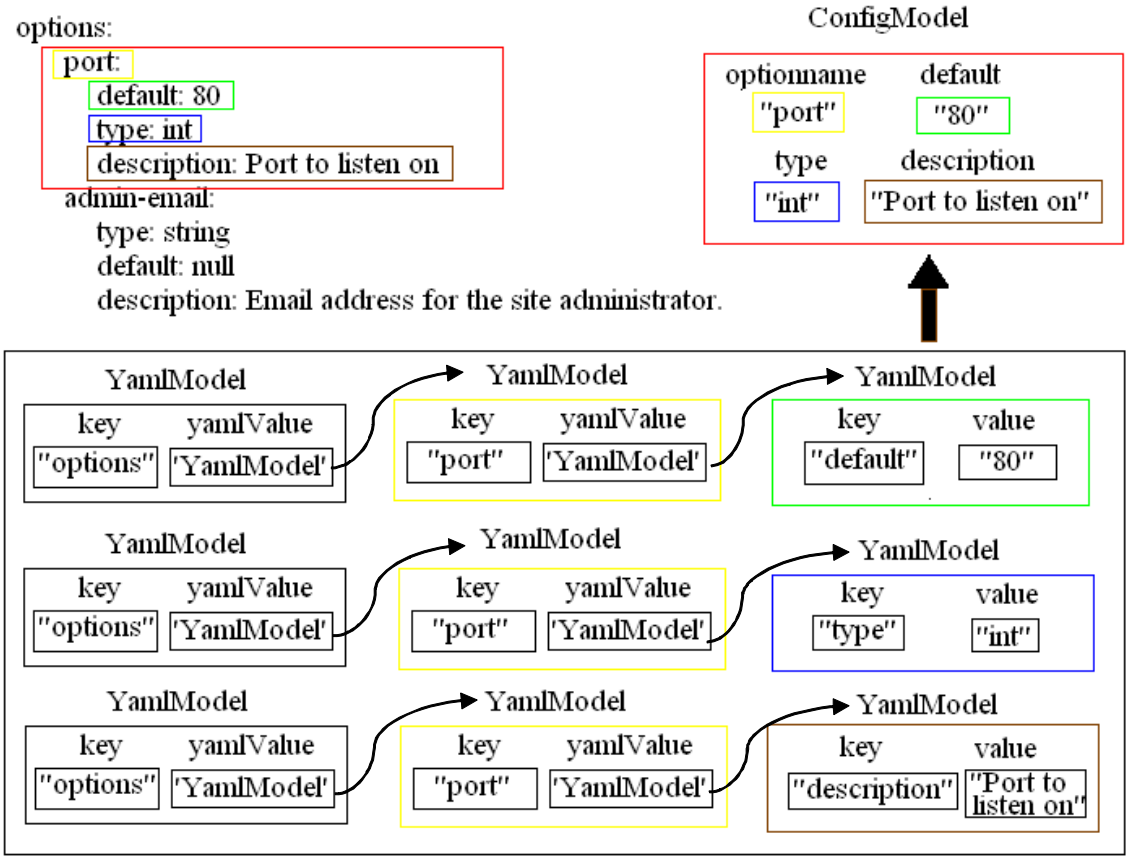

Abbildung 4.2: Das Speichern von Informationen in einer "config.yaml" Datei

4.4.5.2 Methode "elementsImpl"

Die Funktion dieser Methode ist, durch ein YamlModelList-Objekt alle YamlModel-Objekte zu bekommen und die entsprechenden Elemente zu generieren. Die Methode benötigt zwei Eingabeparameter: "name" und "yml".Der Parameter "name" beschreibt den Namen des Service und wird als Paramter an die Methoden "rootImpl" und "complexTypeImpl" weitergegeben. Der Parameter "yml" ist ein Objekt der Klasse "YamlModelList". Durch das YamlModelList-Objekt kann man für jede Option ein ConfigModel-Objekt erzeugen, das als ein Parameter auf die Methode "elementImpl"

übertragen wird. Als Ausgabe liefert diese Methode ein XsdElementsModel-Objekt, in dem alle generierten Elemente gespeichert werden.

4.4.6 Klasse "XsdGenerator"

Die Funktion dieser Klasse ist, durch die oben generierten Elemente das endgültige XSD-Dokument zu erzeugen. Als Erstes wird in dem Konstruktor ein XsdElementsModel-Objekt empfangen, in dem alle Elemente für das XSD-Dokument gespeichert sind. Dann wird die Methode "generator" zum Generieren des XSD-Dokument definiert.

4.4.6.1 Methode "generator"

Die Funktion dieser Methode ist, ein XSD-Dokument zu erstellen, das Root-Element und seine Kindelemente zu diesem Dokument hinzuzufügen und schließlich das Dokument auszugeben. Die Implementierung dieser Methode ist ähnlich wie die Methode der Klasse "XmlGenerator".

5 Zusammenfassung und Ausblick

In dieser Studienarbeit wurde gezeigt, wie eine automatische Prozedur entwickelt werden kann, mit der Artefakte, die von der Juju-Community als „Charms“ veröffentlicht wurden, zu TOSCA-Node-Types konvertiert werden können. Node-Types gehören zu den wichtigsten Bausteinen in TOSCA um Service-Templates und damit Vorlagen für Cloud-Services zu erstellen.

In Kapitel 3.2 wurden die hinter der Prozedur stehenden Konzepte erläutert und es wurde beschrieben, aus welchen Funktionseinheiten die Prozedur besteht. Dabei wurden sowohl der Ablauf der Prozedur sowie die Interaktion ihrer Komponenten mittels eines Sequenzdiagramms in Abbildung 3.3 dargestellt. In Kapitel 4 wurde die Implementierung der wichtigsten Methoden und Klassen der Prozedur besprochen. Besonders ausführlich wurden die Methoden zum Generieren der Elemente des TOSCA-Defnitions-Dokument in Kapitel 4.4.2 beschrieben.

In Abbildung 2.6 wurde gezeigt, dass Topology-Templates und Pläne die zentralen Elemente eines Service-Template sind. Ein Topology-Template besteht aus Node-Templates und Relationship-Templates. Diese Arbeit beschäftigte sich ausschließlich mit der Generierung von Node-Types, die den Typ eines oder mehrerer Node-Templates definieren. Eine Möglichkeit für zukünftige Arbeiten ist die Erzeugung von entsprechenden Relationship-Types, die den Typ eines oder mehrerer Relationship-Templates definieren. Außerdem sollte untersucht werden, wie die hier beschriebene Prozedur auf weitere ähnliche Werkzeuge und Artefakte übertragen werden kann. Ein möglicher, im Umfeld des Cloud Computing ebenfalls etablierter Kandidat wäre bspw. Chef [2].

Literaturverzeichnis

Alle Weblinks wurden das letzte Mal am 22.04.2013 geprüft.

[1] Canonical Juju: https://juju.ubuntu.com

[2] Opscode Chef: http://www.opscode.com/chef

[3] TOSCA Committee: http://www.tosca-open.org

[4] Juju-Community: https://juju.ubuntu.com/community/

[5] Juju Documentation: Charms, https://juju.ubuntu.com/docs/charm.html

[6] Juju Documentation: Writing a Charm, https://juju.ubuntu.com/docs/write-charm.html

[7] Juju Documentation: Service Configuration, https://juju.ubuntu.com/docs/service-config.html

[8] Mell, Grance: The NIST Definition of Cloud Computing National Institute of Standards and Technology, NIST, 2011

[9] Baun, Kunze, Nimis, Tai: Cloud Computing - Web-basierte dynamische IT-Services, 2. Aufl. ed., Heidelberg, Dordrecht, London, New York: Springer-Verlag, 2011.

[10] Binz, Breiter, Leymann, Spatzier: Portable Cloud Services Using TOSCA. In: Internet Computing, IEEE, IEEE, 2012, 16, 80-85

[11] Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd05/TOSCA-v1.0-csd05.html

[12] Business Process Model and Notation (BPMN) Version 2.0, Object Management Group specification, Jan. 2011.

[13] Web Services Business Process Execution Language (BPEL) Version 2.0., OASIS specification, 2007.

[14] W3C. Web Services Description Language (WSDL) 1.1. [Online] March 15, 2001. http://www.w3.org/TR/wsdl.

[15] Kopp, Binz, Breitenbücher, Leymann: BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In: Mendling, Jan (Hrsg); Weidlich, Matthias (Hrsg): 4th International Workshop on the Business Process Model and Notation, 2012.

[16] Leymann, Fehling, Mietzner, Nowak, Dustdar: Moving Applications to the Cloud: An Approach based on Application Model Enrichment. In: International Journal of Cooperative Information Systems (IJCIS). Vol. 20(3), World Scientific, 2011.

[17] Java SE 6 Documentation: http://docs.oracle.com/javase/6/docs/

[18] Snakeyaml: http://code.google.com/p/snakeyaml/

[19] Dom4j: http://dom4j.sourceforge.net/

[20] Booch, Rumbaugh, Jacobson: Unified Modeling Language User Guide, Second Edition; Addison-Wesley Professional;

[21] Horstmann, Cornell: Core Java, Volume I--Fundamentals (8th Edition).

[22] Java SE 6 Documentation: Package java.util, http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html

[23] Wettinger, Behrendt, Binz, Breitenbücher, Breiter, Leymann, Moser, Schwertle, Spatzier: Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER), 2013

[24] Java SE 6 Documentation: Package java.util.zip, http://docs.oracle.com/javase/6/docs/api/java/util/zip/package-summary.html

[25] Java SE 6 Documentation: Package java.io, http://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html

[26] Dom4j 1.6.1 API: Package org.dom4j, http://dom4j.sourceforge.net/dom4j-1.6.1/apidocs/

[27] SnakeYAML Documentation: Loading YAML, http://code.google.com/p/snakeyaml/wiki/Documentation#Loading_YAML

[28] YAML: http://www.yaml.org/

[29] Drupal: http://drupal.org

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben. Wörtliche und sinngemäße Übernahmen aus anderen Quellen habe ich nach bestem Wissen und Gewissen als solche kenntlich gemacht.

Stuttgart, den 25. April 2013 _____