Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Diplomarbeit Nr. 3605

TOSCA Kompatibilitäts- und Compliance-Testumgebung

Theo Aouidet

Studiengang: Informatik

Prüfer: Prof. Dr.-Ing. habil. Bernhard Mitschang

Betreuer: Dr. rer. net. Matthias Wieland

begonnen am: 18.11.2013

beendet am: 22.05.2014

CR-Klassifikation: D.2.5, H.3.5

Kurzfassung

Im durch das BMWi geförderten Projekt CLOUDECYCLE wird Cloud-Betreibern ermöglicht, Dienste bei garantierter Sicherheit und Compliance (d.h. Einhaltung von Gesetzen, Richtlinien und Datenschutzvorgaben) kosteneffizient und skalierbar für Kunden des Mittelstands und der öffentlichen Verwaltung bereitzustellen. Hierbei sichert CLOUDCYCLE den gesamten Lebenszyklus der Dienste ab und stellt die Sicherheitsgarantien von Diensten sowie deren Anforderungen an die Laufzeitumgebung zu spezifizieren, Dienste vollautomatisch bereitzustellen und zu betreiben, sowie zwischen Betreibern zu migrieren, sicher.

Ziel meiner Diplomarbeit ist es eine Testumgebung für die TOSCA-Engine (OpenTOSCA) zu erstellen. Dadurch soll es möglich werden zu testen, inwieweit die OpenTOSCA den TOSCA-Standard erfüllt.
Zudem wird eine Vergleichbarkeit verschiedener Laufzeitumgebungen für die Zukunft realisierbar [Wie13].

Inhaltsverzeichnis

Kurzfassung **2**
Inhaltsverzeichnis **3**
Abbildungsverzeichnis **5**
Verzeichnis der Listings **7**
1 Einleitung **8**
1.1 Aufbau der Arbeit 8
2 Verwandte Arbeiten **9**
3 Grundlagen **10**
3.1 Cloud Computing 10
3.2 TOSCA 11
3.3 OpenTOSCA 13
3.4 CSAR 14
3.5 WS-BPEL 15
3.6 Winery 16
3.7 Das imperative Modell 17
3.8 Dynamischer Software-Test 18
3.8.1 White-Box-Verfahren 18
3.8.2 Black-Box-Verfahren 18
4 Testkonzept **20**
4.1 Teststrategie 20
4.2 Erste Testphase: Komponententest 20
4.2.1 Service Template Test 21
4.2.2 Artefakte Test 21
4.2.3 Plan Test 22
4.3 Zweite Testphase: Integrationstest 22
4.3.1 Testfälle erzeugen 22
4.3.2 Integrationstest 23
5 Implementierung **25**
5.1 Testfälle Erzeugung 25
5.1.1 Die TOSCA-Spezifikation 25
5.1.1.1 Die Service Templates 25
5.1.2 Artefakte 28
5.1.2.1 Web-Service 31
5.1.3 Pläne 36
5.1.3.1 Modellierungstool 38

5.1.3.2 WSO2 BPS 39
5.1.4 Erstellung der CSARs 41
5.2 Die Testumgebung bereitstellen 43
5.3 Ausführung der Testfälle 45
5.4 Analyse der Testdurchführung 51
6 Zusammenfassung und Ausblick 52
Literaturverzeichnis 56
Anhang 58
Erklärung 59

Abbildungsverzeichnis

Abbildung 1: Cloud Computing [COM11] ... 11

Abbildung 2: TOSCA- Meta-Modell ... 12

Abbildung 3: OpenTOSCA [Ope13] ... 13

Abbildung 4: OpenTOSCA Architektur [Ope13] ... 14

Abbildung 5: CSAR Ordnerstruktur ... 15

Abbildung 6: Winery Komponenten [KBBL13] ... 16

Abbildung 7: Topology-Modeler [KBBL13] ... 17

Abbildung 8: Element Manager [KBBL13] ... 17

Abbildung 9: Service Templates bzw. Testfälle aus der Spezifikation ableiten ... 21

Abbildung 10: Mögliche Artefakte Tests ... 22

Abbildung 11: Plan Test ... 22

Abbildung 12: CSAR-Dateien (-Files) erzeugen ... 23

Abbildung 13: Integrationstest ... 24

Abbildung 14: Service Template [TOSCA13] ... 26

Abbildung 15: Graphische Modellierung eines Service Templates mit Winery ... 28

Abbildung 16: Implementation/ Deployment Artefakte ... 30

Abbildung 17: WSDL-Graph des Web-Services ... 34

Abbildung 18: Test des Web-Services mit dem Web Services Explorer ... 34

Abbildung 19: Erfolgreiche Ausgabe auf der Konsole ... 35

Abbildung 20: Web-Service-Test mit der SoapUI ... 35

Abbildung 21: Graphische Modellierung des XML-Codes von Listing 3 mit BPEL-Designer ... 39

Abbildung 22: WSO2 BPS ... 40

Abbildung 23: Try-It-Funktion des WSO2 BPS ... 41

Abbildung 24: CSAR-Ordnerstrukur manuell (a) und mit Winery (b) erzeugt 42

Abbildung 25: Die Testumgebung 44

Abbildung 26: Testfall 1 erfolgreich geladen 46

Abbildung 27: Testfall 18 erfolgreich geladen 46

Abbildung 28: Inhalt vom geladenen Testfall 18 47

Abbildung 29: Testfall 18 im Container API der OpenTOSCA erfolgreich installiert 47

Abbildung 30: eine CSAR-Datei aus der Testfallgruppe „Deployment Testfälle“ erfolgreich geladen 48

Abbildung 31: Die CSAR ExampleWS erfolgreich im Container API der OpenTOSCA installiert 48

Abbildung 32: Inhalt der CSAR ExampleWS im Container API der OpenTOSCA 49

Abbildung 33: Implementation Artefakt Example_WS erfolgreich in die Tomcat installiert. 49

Abbildung 34: ausgeführter Plan in WSO2 BPS 50

Abbildung 35: die neu erzeugte Instanz in WSO2 BPS 50

Abbildung 36: Regressionstest (Automation Allgemein) 53

Abbildung 37: Automation der Testumgebung 54

Abbildung 38: Struktur der mitgelieferten CD 58

Verzeichnis der Listings

Listing 1: Service Template XML-Code 28

Listing 2: WSDL-Quellcode eines Web-Services 33

Listing 3: XML-Code Ausschnitt aus einem Testfall in BPEL-Designer 38

1 Einleitung

Es ist durchaus eine Tatsache, dass Cloud Computing heutzutage die IT-Welt entscheidend revolutioniert hat und große Veränderungen in IT-Unternehmen bewirkt hat. Cloud Computing bietet Unternehmen die Möglichkeit ihre IT-Infrastruktur voll oder teilweise in die Cloud zu migrieren. So werden zum einen Kosten für Ressourcen gespart und zum anderen können Vorteile wie Skalierbarkeit, Flexibilität oder Verfügbarkeit in der Cloud genutzt werden.

Mit TOSCA ist eine Spezifikation entstanden, die das Ausführen und Verwalten von Cloud-Anwendungen durch ein Meta-Modell möglich macht. Cloud-Anwendungen werden in TOSCA durch TOSCA-Container-Files, auch Cloud-Service-Archive (CSAR) genannt, realisiert. Ein TOSCA-Container soll eine Laufzeitumgebung sein, die zur Ausführung und zum Management von CSARs dient. Diese Laufzeitumgebung muss die TOSCA-Spezifikation interpretieren und erfüllen können.

Der Aufwand neue TOSCA-Laufzeitumgebungen immer wieder auf ihre Kompatibilität und Compliance gemäß dem TOSCA-Standard zu überprüfen, ist hoch und nimmt sehr viel Zeit in Anspruch. Aus diesem Grund ist es das Ziel dieser Arbeit, eine Testumgebung zu entwickeln, mit der Entwickler bzw. Tester in der Lage sind, verschiedene TOSCA-Container gemäß der TOSCA-Spezifikation auf Kompatibilität testen zu können. Durch diese Testumgebung wird zum einen die Möglichkeit gegeben, Fehler zu entdecken und gegebenenfalls zu beheben und zum anderen wird der Vergleich mehrerer Laufzeitumgebungen ermöglicht.

Speziell in dieser Diplomarbeit wird die TOSCA-Laufzeitumgebung OpenTOSCA als Testobjekt dienen. Diese Browser-basierte TOSCA-Laufzeitumgebung (OpenTOSCA), die zur Ausführung von TOSCA-basierten Anwendungen (CSARs) benutzt wird, wird die Testumgebung durchlaufen müssen. Die Ergebnisse der Testdurchführung werden dann dokumentiert.

1.1 Aufbau der Arbeit

Die vorliegende Arbeit ist in sechs Kapiteln unterteilt:
Kapitel 1 beginnt mit einer Einführung in das Themengebiet zu dieser Arbeit.
In Kapitel 2 werden verwandte Arbeiten bzw. verwendete Materialien vorgestellt.
Darauf folgt in Kapitel 3 eine Erläuterung der Begrifflichkeiten, die für das Verständnis dieser Arbeit relevant sind.
Im Anschluss wird in Kapitel 4 das Testkonzept detailliert präsentiert.
Im darauffolgenden Kapitel 5 wird zuerst die Entwicklung der Testfälle beschrieben, danach die Durchführung der Tests und zum Schluss eine Analyse der Testdurchführung. Im Grunde wird in diesem Kapitel die gesamte Implementierung der Testumgebung vorgestellt.
Abschließend werden in Kapitel 6 die gewonnenen Erkenntnisse zusammengefasst, Möglichkeiten und Grenzen der realisierten Lösung aufgezeigt und es wird einen Ausblick auf zukünftige Arbeiten gegeben.

2 Verwandte Arbeiten

Da TOSCA vor nicht allzu langer Zeit eingeführt wurde, haben sich bisher keine Arbeiten mit dem Testen der TOSCA-Engine beschäftigt. Die vorliegende Diplomarbeit sollte die Erste sein, die sich diesem Thema widmet.
In diesem Kapitel werden Arbeiten, die für den Einstieg in die TOSCA Welt sowie für die Bearbeitung dieser Diplomarbeit selbst als Hilfe genommen wurden, vorgestellt.

Von besonderer Bedeutung sind die TOSCA-Spezifikation [TOSCA13], die TOSCA-Primer [TOS13] und das TOSCA-XML-Schema [OAS12] Sie sind feste Bestandteile dieser Diplomarbeit.
Zu Beginn diente die CLOUDCYCLE-Plattform [CC12], die Veröffentlichungen sowie zahlreiche interessante Beiträge im Zusammenhang mit TOSCA anbietet, als Einstieg in die TOSCA-Welt.
Relevant für diese Diplomarbeit waren ebenfalls die Bücher „Basiswissen Softwaretest“ [SL11] und „Praxiswissen Softwaretest“ [SLRW07], die als Grundlage für die Komponenten-und Integrationstests verwendet wurde.
Im nächsten Schritt wurde für eine grundlegende Einarbeitung in XML die dreiteilige Übersetzung des W3C XML-Schemas zur Hilfe genommen: Einführung [XML01a], Struktur [XML01b] und Datentypen [XML01c].
Des Weiteren wurden die Diplomarbeit von David Schumm [SD08] und die Primer [BPE07] als Einstiegshilfe in die grafische Modellierung genommen.
Abschließend wurde die Plattform [Ope12], welche eine umfassende Einführung in die Openstack Software bietet, genutzt.

3 Grundlagen

In diesem Kapitel werden zentrale und grundlegende Begriffe, auf der sich diese Arbeit basiert und welche in den weiteren Kapiteln als vorausgesetzt angenommen werden, erläutert.

3.1 Cloud Computing

Cloud bedeutet Wolke auf Deutsch, Computing bedeutet Rechnen. Zusammen heißt es in etwa Rechnen in der Wolke. Das Cloud steht aus Nutzersicht für die Undurchsichtigkeit und die Komplexität der zur Verfügung gestellten Ressourcen, wie zum Beispiel, wo sie sich genau befinden, was in ihnen geschieht oder wie die Dienste angeboten werden.

Ziel des Cloud Computing ist es, Dienste möglichst transparent anzubieten. Dabei spielt die Technologie und die ganze Architektur für den Nutzer bzw. den Kunden eine nebensächliche Rolle. Der Fokus liegt vielmehr auf der Bedienbarkeit für den Benutzer. Das Konzept sieht vor, den Nutzern verschiedene Ressourcen mithilfe von definierten technischen Schnittstellen zugänglich zu machen. So sollte sich der Kunde im Idealfall nicht mit der Administration auseinandersetzten müssen. Cloud Computing (siehe Abb. 1) ist eine Art Plattform zur Beschreibung von Technologien, die den Kunden Lösungen zu Verfügung stellt. Die Nutzer müssen Dienste mieten, um sie dann nutzen zu dürfen. Im Cloud Computing bezahlen Kunden nur soviel wie sie tatsächlich genutzt haben. Ein weiterer Vorteil des Cloud Computing ist die geografische Unabhängigkeit der Nutzer. Sie können nämlich weltweit Zugriff haben, dazu brauchen sie lediglich einen Internetzugang. Hier spricht man von einer „Public Cloud“ als Liefermodell.

Das Konzept des Cloud Computing beinhaltet den gesamten Umfang der Informationstechnik und wird, wie die National Institute of Standard and Technology (NIST) definiert hat, in drei Teilen kategorisiert: Infrastruktur (IaaS), Plattformen (PaaS) und Softwares (SaaS).
So können in IaaS Teile der IT-Infrastruktur etwa wie Rechenzentrum oder Datenspeicher nicht mehr vom Nutzer lokal betrieben, sondern werden von anderen geografisch verteilten Anbietern übernommen. Vereinfacht ausgedrückt, mietet sich der Nutzer diese Dienste, die sich dann in der Cloud und nicht mehr auf dem lokalen Rechner befinden. Diese Dienste können über ein Netzwerk wie das Internet („Public Cloud“ genannt) oder über ein firmeninternes Intranet („Private Cloud“ genannt), geliefert werden.
Mit der PaaS werden diverse Plattformen von Anbietern zur Verfügung gestellt, die den Nutzern die Möglichkeit geben, eigene Software-Anwendungen zu entwickeln oder auch Anwendungen in Laufzeitumgebungen ausführen zu lassen. Die Anbieter müssen die Verfügbarkeit sowie die Sicherheit dieser Plattformen gewährleisten und sind für die Wartung dieser zuständig.
Als Letztes können Softwares und Anwendungen jeglicher Art als SaaS in der Cloud angeboten werden. Nutzer können diese speziellen Softwares, die von den Anbietern auf ihrer eigenen Infrastruktur verwaltet werden, nutzen [NAT11].

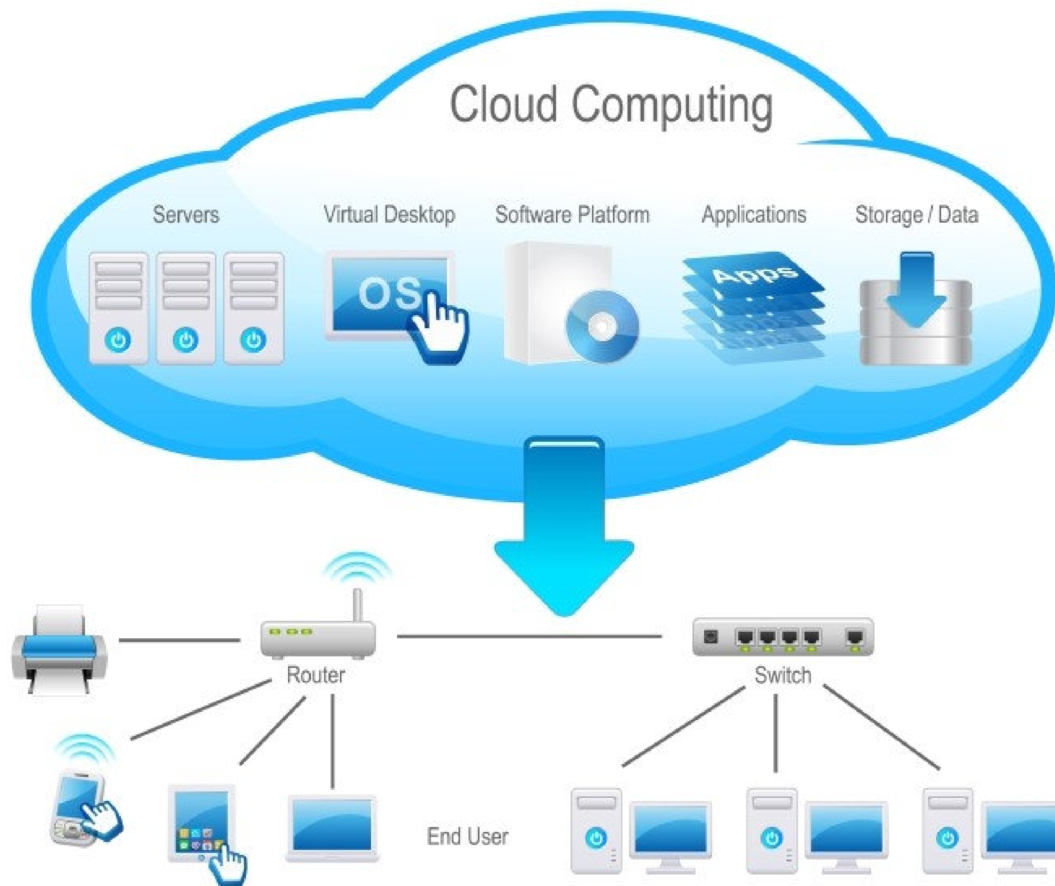

Abbildung 1: Cloud Computing [COM11]

3.2 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) wurde im Rahmen eines OASIS-Projekts in Zusammenarbeit mehrerer Industriefirmen wie IBM oder SAP entwickelt. Dabei war das Ziel einen offenen Standard, der die Portabilität und Interoperabilität von Cloud-Anwendungen und -Diensten gewährleistet, anzubieten.
Portabilität im Sinne von TOSCA bedeutet, dass Definitionen und Strukturen von Diensten auch von anderen Anbietern, die den TOSCA-Standard nutzen, verstanden und interpretiert werden können. Damit die Portabilität tatsächlich funktioniert, muss sichergestellt werden, dass alle Funktionalitäten innerhalb einer TOSCA-Container-Datei von der Laufzeitumgebung gestützt werden.

Das Ziel von TOSCA ist es, Abhängigkeiten von einzelnen Cloud-Anwendungen-Anbietern (Lock-Ins) zu verhindern. Die Spezifikation bietet einen Standard, der das Format von Cloud-Anwendungen regelt, an. Dadurch können Service Templates anbieterunabhängig interpretiert werden. Die Struktur und das Verhalten von einer Cloud-Anwendung stehen anderen Containern zur Verfügung und können, auch wenn die Anwendung ursprünglich für einen anderen Zweck entwickelt wurde, durch sie interpretiert und genutzt werden. So wird den Entwicklern die Möglichkeit gegeben, durch das Kombinieren von verschiedenen Anwendungen bzw. Komponenten anderer Anbieter, weitaus komplexe Dienste zu entwickeln. Damit wird der zweite Aspekt, nämlich die Interoperabilität, sichergestellt.

Mit dieser Unabhängigkeit und der Interoperabilität bietet die TOSCA-Spezifikation einen hohen Grad an Flexibilität, die es möglich macht, den Kunden einen breiten Pool von standardisierten Cloud-Diensten zur Verfügung zu stellen, aus dem sie sich je nach Präferenzen ihre eigenen Lösungen zusammenstellen können.
Des Weiteren definiert TOSCA ein Meta-Modell, das die Beschreibung von IT-Diensten erlaubt. Dieses Meta-Modell ermöglicht die Entwicklung eines Modells für die Topologie von Cloud-Diensten. Das TOSCA-Meta-Modell beinhaltet unter anderem Typen, Templates und Pläne (siehe Abb.2).
Der gesamte Lebenszyklus einer Cloud-Anwendung, von der Entstehung, über das Management bis hin zur Terminierung, kann mit TOSCA definiert werden. Die komplette Struktur wird durch das Topology Template bestimmt. Zum Verwalten und Ausführen der Anwendung werden Pläne eingesetzt. Zur Repräsentation eines TOSCA-Modells kommt das Datenformat XML zum Einsatz.
Abschließend, es wird eine Laufzeitumgebung gebraucht, um das Cloud-Service-Archive (kurz CSAR), das eine Cloud-Anwendung darstellen kann, in TOSCA ausführen zu können. In der TOSCA-Spezifikation wird die Laufzeitumgebung TOSCA-Container genannt [TOSCA13].

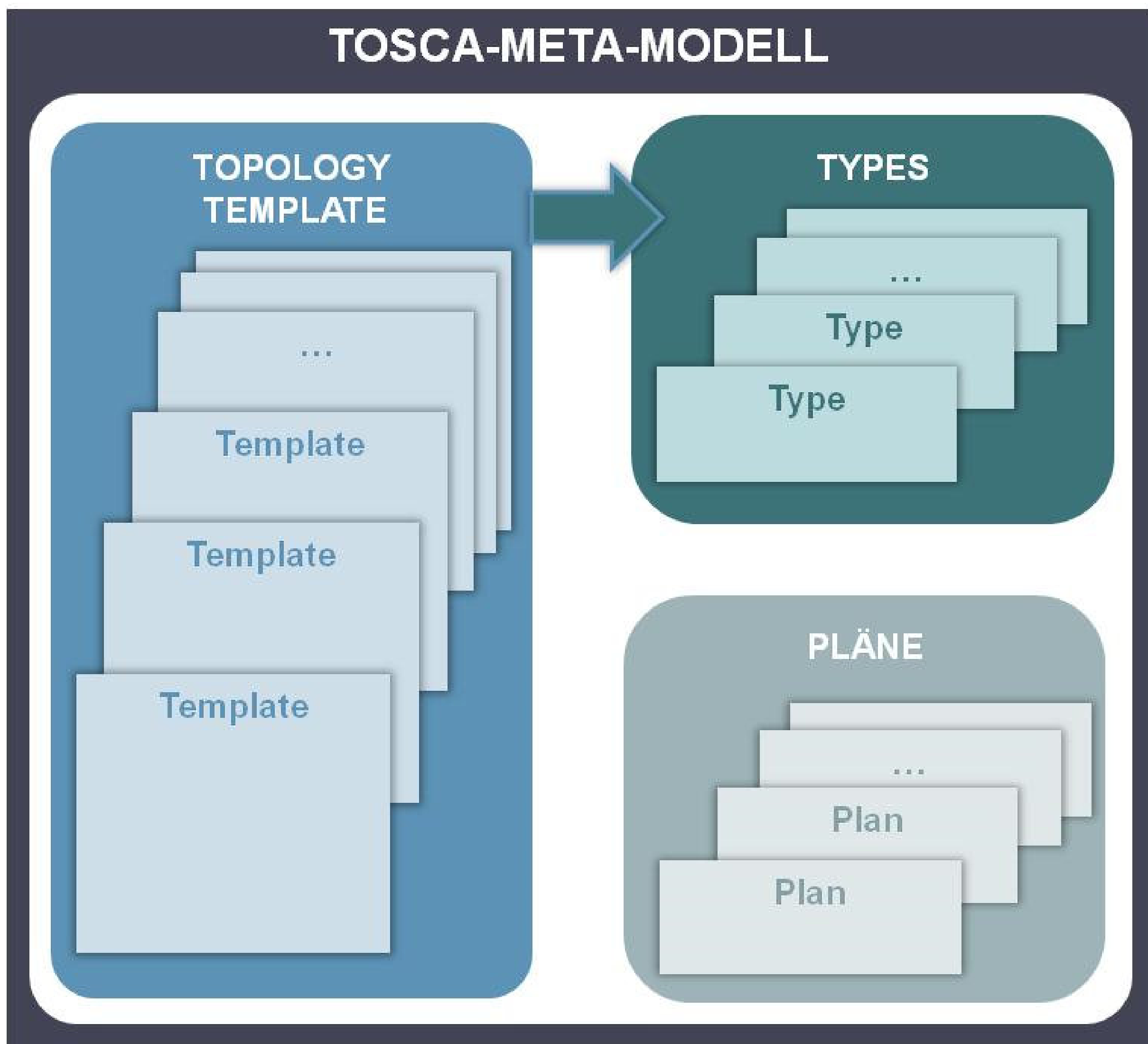

Abbildung 2: TOSCA- Meta-Modell

3.3 OpenTOSCA

OpenTOSCA ist eine Browser-basierte Open-Source-Implementierung eines TOSCA-Containers (siehe Abb.3), die an der Universität Stuttgart entwickelt wurde. Wie oben schon erwähnt, dient dieser Container als Laufzeitumgebung zur Ausführung und Verwaltung von Cloud-Anwendungen gemäß dem TOSCA-Standard [Ope13].

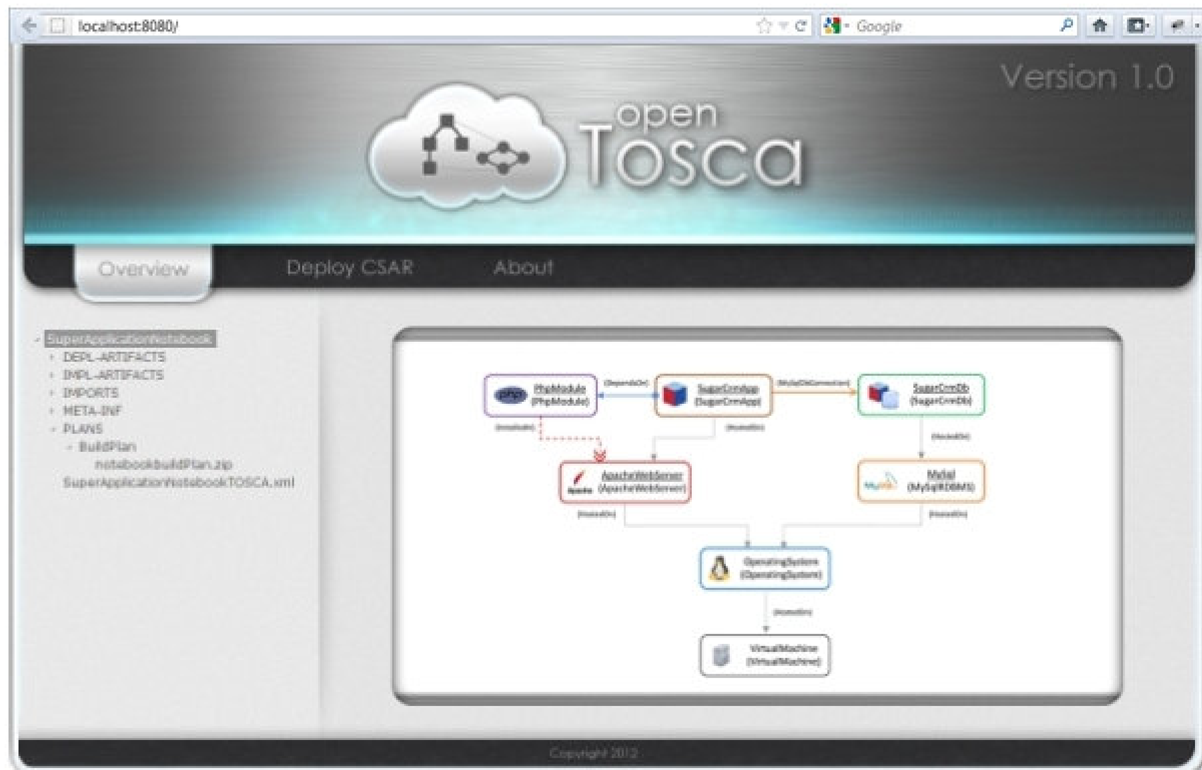

Abbildung 3: OpenTOSCA [Ope13]

OpenTOSCA unterstützt momentan nur das imperative Modell. Die deklarative Herangehensweise wird für spätere Versionen möglicherweise eingeführt. Als TOSCA-Container sollte OpenTOSCA in der Lage sein, TOSCA-Container-Dateien (CSARs) zu laden, ausführen und verwalten. Diese Dateien haben ein vorgegebenes Format und beinhalten mehrere Ordner.

Die Architektur von OpenTOSCA (Abb. 4) besteht grundsätzlich aus einer Implementation Artifacts Engine (IAE), die alle in der TOSCA-Container-Datei enthaltenen Implementation Artifacts automatisch ausführt. Die Plan Engine stellt die Pläne, die für das Management von Anwendungen verantwortlich sind, bereit. Dabei müssen Pläne an die zugehörigen Management Operationen, welche von Implementation Artifacts implementiert werden, gebunden werden. Die dritte Komponente der OpenTOSCA ist der Controller. Dieser steuert alle anderen Komponenten und stellt eine Schnittstelle zur Verwaltung, Installierung und Terminierung von Container-Dateien, zur Verfügung. Weitere Komponenten wie Datenspeicher-, Datenzugriffs- und Verwaltungskomponenten, die für die Datenhaltung der Instanz- und Modellverwaltung zuständig sind, sind in der Architektur ebenfalls zu finden. Das Zusammenspiel dieser Komponenten bietet die Funktionalität [Ope13].

In dieser Diplomarbeit wird die OpenTOSCA-Engine das Testobjekt sein, was nach Kompatibilität und Compliance getestet wird. Damit sollte überprüft werden, inwieweit sie den TOSCA-Standard erfüllt.

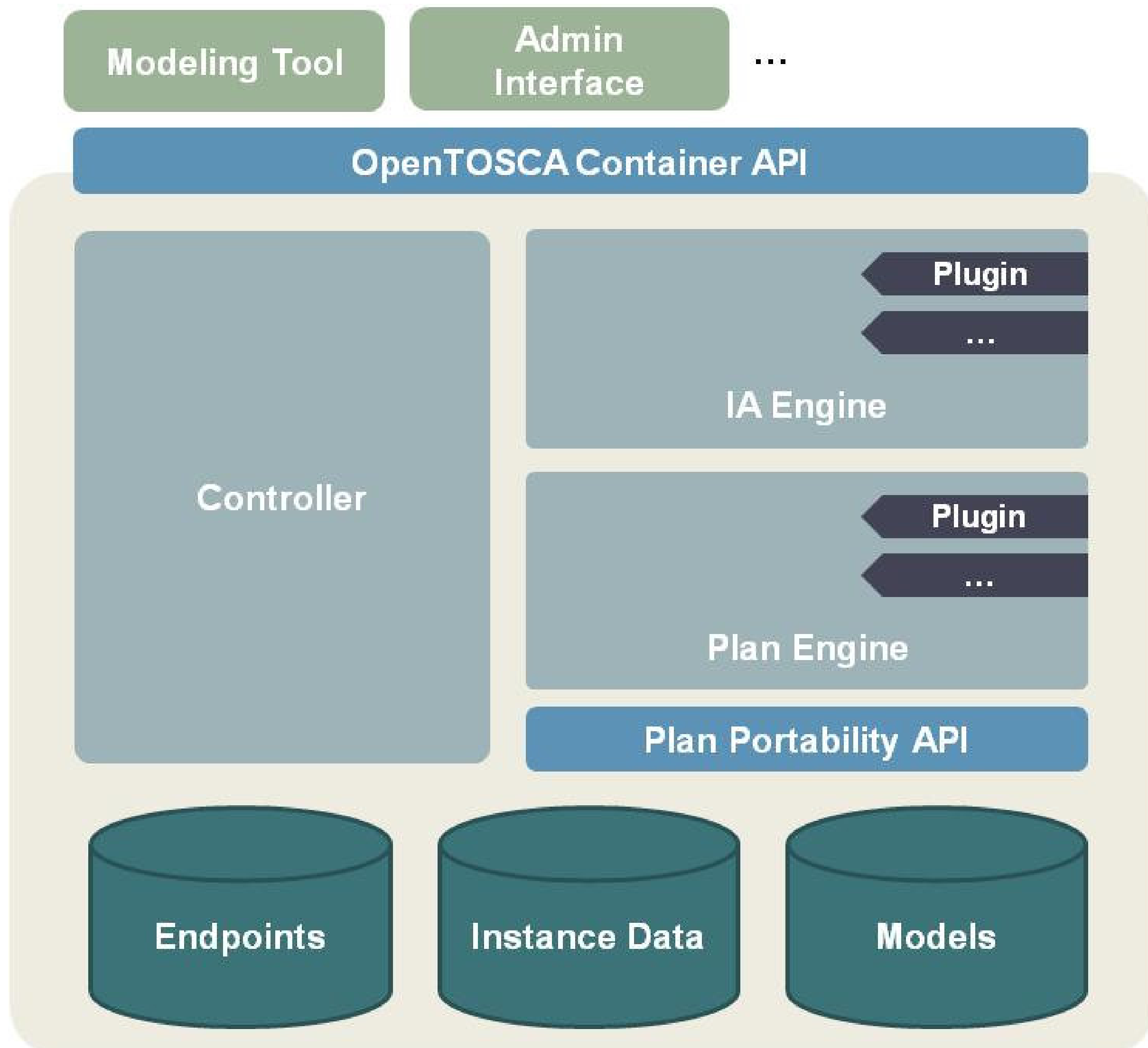

Abbildung 4: OpenTOSCA Architektur [Ope13]

3.4 CSAR

Cloud-Service-Archive oder kurz CSAR genannt, ist die TOSCA-Container-Datei, die gemäß dem TOSCA-Standard eine vorgegebene Struktur (siehe Abb. 5) mindestens haben muss und eine TOSCA-basierte Cloud-Anwendung darstellt. Das Archiv besteht aus mehreren Ordnern, die in einer Zip-Datei zusammengepackt sind. Diese Zip-Datei beinhaltet alle Dateien wie beispielsweise Service Templates, Software Artefakte und Management Pläne, die zur Instanziierung des Services benötigt werden. Auf Basis dieser Dateien können mithilfe der Pläne Instanzen erzeugt und verwaltet werden [Ope13].

Für die Diplomarbeit werden mehrere CSAR-Dateien entwickelt, die dann die Testfälle der Testumgebung bilden werden. Sie werden in die OpenTOSCA geladen und ausgeführt, um die Kompatibilität und Compliance der Laufzeitumgebung OpenTOSCA zu überprüfen. Diese CSARs können wiederum für das Testen weiterer Laufzeitumgebungen eingesetzt werden.

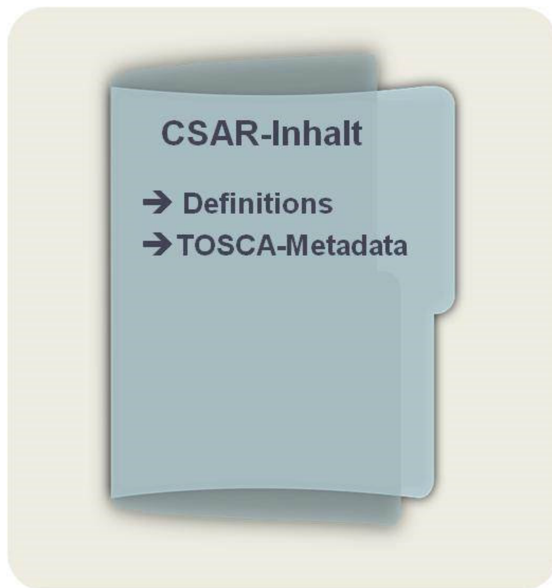

Abbildung 5: CSAR Ordnerstruktur

Abbildung 5 zeigt die CSAR-Ordnerstruktur. Das CSAR-Verzeichnis muss mindestens die Unterordner „Definitions“ und „TOSCA-Metadata“ beinhalten. Im Unterordner „Definitions“ muss sich mindestens ein Service Template befinden. Im Unterordner „TOSCA-Metadata“ hingegen ist die „TOSCA.meta“ enthalten, in der Meta-Informationen über den CSAR zu finden sind.

3.5 WS-BPEL

Web Services Business Process Execution Language (WS-BPEL) ist eine weitverbreitete Sprache zur Beschreibung und Orchestrierung von Web-basierten Geschäftsprozessen, mithilfe von Workflow-Plänen. Diese XML-basierte Spezifikation ermöglicht das Zusammenstellen von unterschiedlichen Services zu einer Komposition, die wiederum einen ausführbaren Geschäftsprozess beschreibt. BPEL ist dafür konzipiert, die Modellierung auf einer hohen abstrakten Ebene möglich zu machen. Mit dieser Abstraktion können komplexe Zusammenhänge festgelegt werden, ohne dabei den Überblick zu verlieren.
Im Grunde existieren bereits zahlreiche Open Source Tools, die genutzt werden können, wie z. B. BPEL-Designer. Diese Eclipse-Anwendung kann zur Unterstützung bei der Erstellung von WS-BPEL Plänen eingesetzt werden. Die daraus entstandenen Pläne können ihrerseits in einem Applikationsserver wie Apache ODE oder WSO2 BPS geladen und dort zur Ausführung gebracht werden. Mit diesen Modellierungswerkzeugen ist es möglich, einzelne Abläufe mithilfe von einem grafischen Editor darzustellen. Damit wird das Erstellen von Geschäftsprozessen sehr transparent gestaltet [OAS07].

3.6 Winery

Winery ist ein Modellierungstool speziell für TOSCA-basierte Anwendungen. Es wurde vom Institut für Architektur von Anwendungssystemen an der Universität Stuttgart entwickelt. Mit dieser Software kann man praktisch eine vollständige Cloud-Anwendung gemäß dem TOSCA-Standard samt Artefakten und Plänen bereitstellen. Nachdem die graphische Modellierung der Topologie abgeschlossen wird, steht die ganze Anwendung als Cloud-Service-Archive (CSAR) zum Exportieren bereit. Sie kann zum Beispiel auf dem Rechner runtergeladen werden und liegt zur Ausführung in die OpenTOSCA wie oben bereits beschrieben vor.

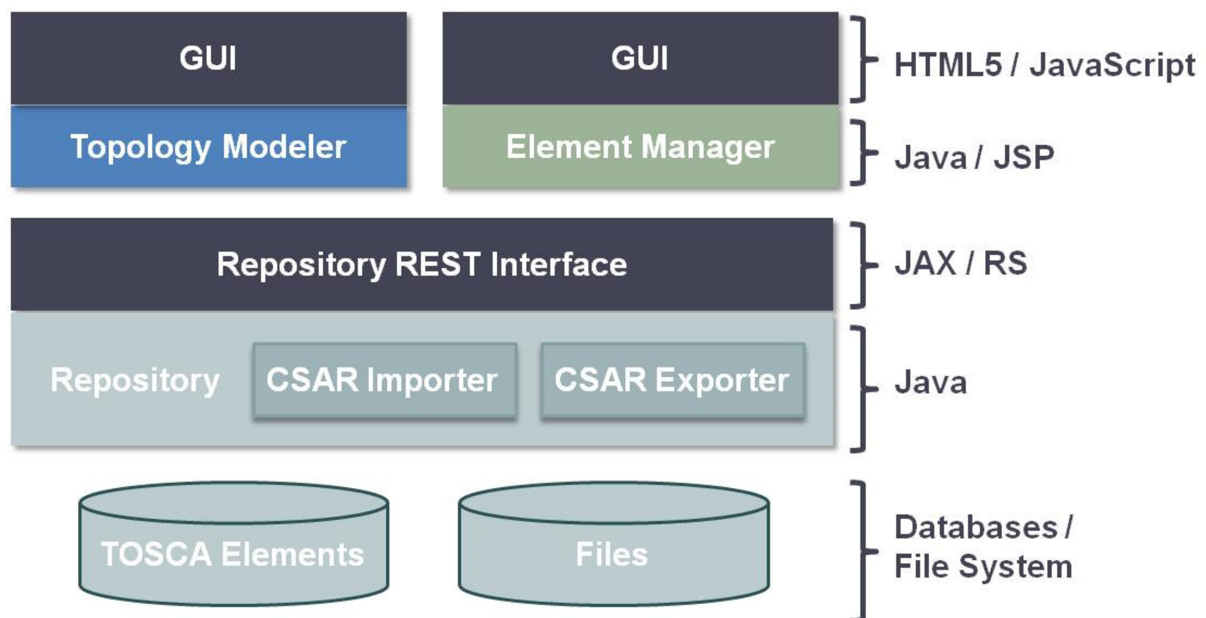

Abbildung 6: Winery Komponenten [KBBL13]

Die Architektur von Winery (siehe Abb. 6) kann im Wesentlichen in drei Bereichen zusammengefasst werden:

- Der Topology-Modeler: Mit diesem wird dem Entwickler die Möglichkeit gegeben, ein Service Template bzw. ein Topology Template, das die gesamte Topologie einer Anwendung, wie es im Anwendungsbeispiel Moodle in Abbildung 7 unten gezeigt wird, grafisch darstellen zu können. Hier werden Node Templates durch abgerundete Rechtecke dargestellt. Darüber hinaus repräsentieren Pfeile, die Node Templates miteinander verbinden, die Relationship Templates.
- Der Element Manager (Abb. 8): Er wird dazu genutzt, um beispielsweise neue erforderliche Typen zu definieren, die für die Erstellung der Topologie eventuell gebraucht werden können. Es gibt aber bereits zahlreiche vordefinierte Typen, die den Entwicklern zur Verfügung stehen.
- Der Repository: Er dient als Speicher für die erstellten und die hochgeladenen Daten, die jederzeit verwendet oder als Cloud-Service-Archive (CSAR) exportiert werden können [KBBL13].

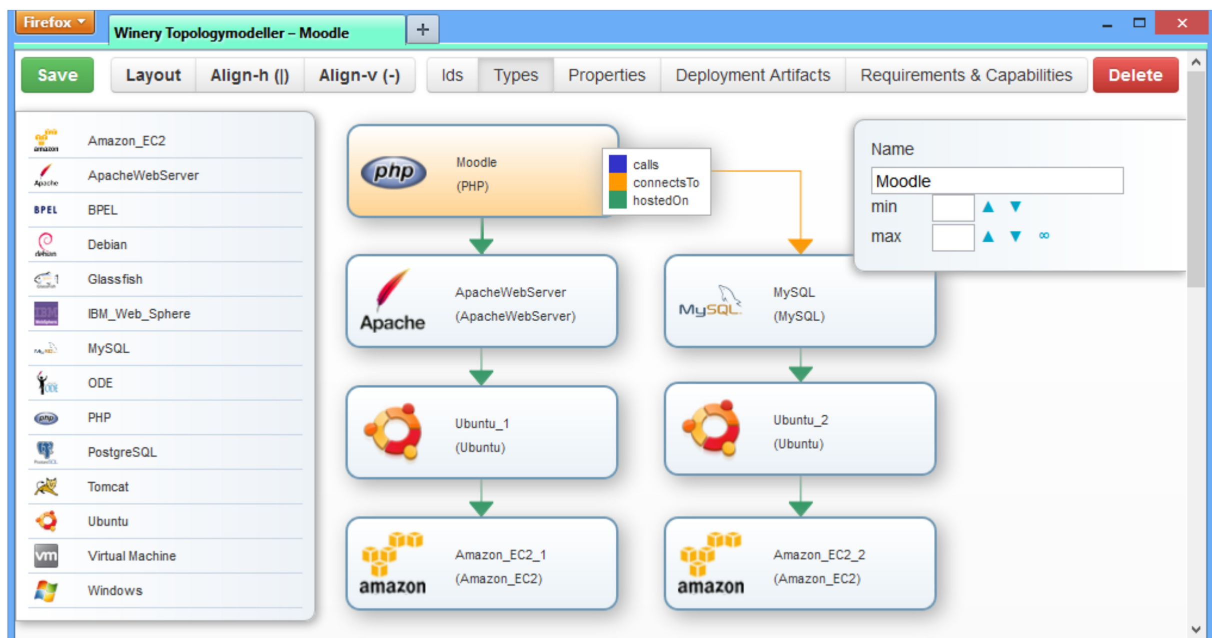

Abbildung 7: Topology-Modeler [KBBL13]

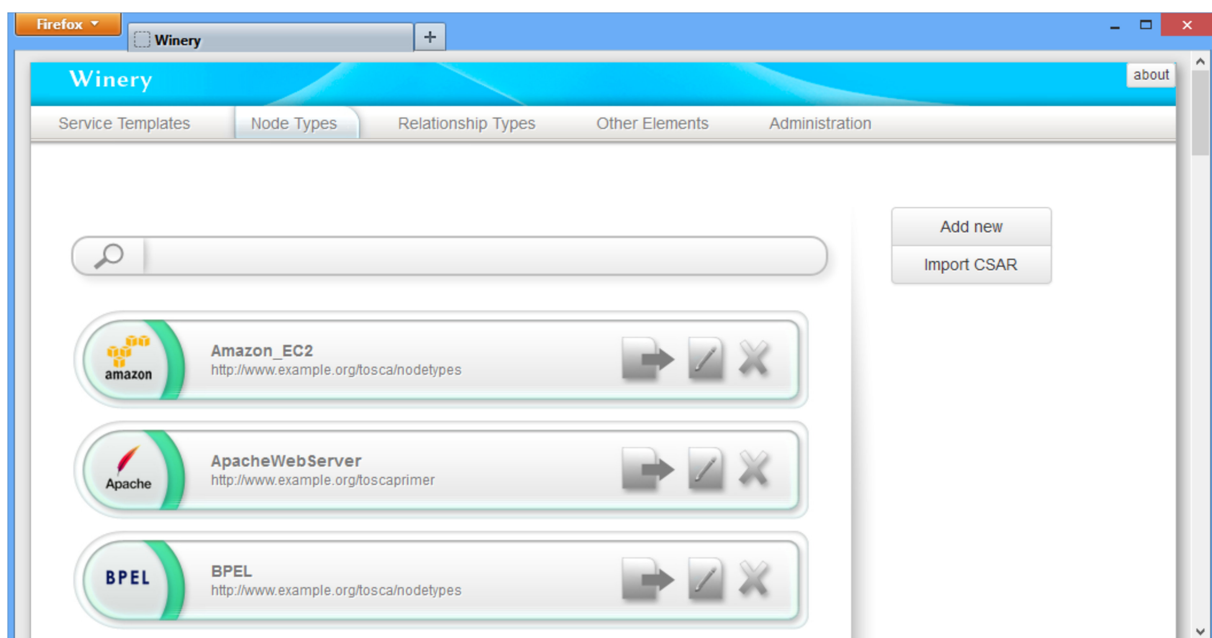

Abbildung 8: Element Manager [KBBL13]

3.7 Das imperative Modell

Das imperative Modell ist ein Programmierungskonzept. Im Programm wird anhand von einer vorgegebenen Reihenfolge bestimmt, was vom Computer genau getan werden soll. Des Weiteren ist es erlaubt, durch eine Reihe von Operationen den Programmzustand zu verändern. Die Idee besteht darin, den Computer als eine klassische Von-Neumann-

Architektur zu betrachten, in dem im Quellcode einer Programmiersprache festgelegt wird, was in welcher Reihenfolge und wie zu tun ist. Im Gegensatz zum deklarativen Modell, bei dem der Entwickler im Quellcode definiert, was das Programm tun muss, aber nicht wie es dies zu tun hat, wird im imperativen Modell genau vorgegeben, wie sich das System anhand von Befehlsfolgen zu verhalten hat. Letztlich ist es wie bei natürlichen Sprachen, bei denen die Bedeutung von Imperativ nichts Anderes ist, als dem Computer zu sagen, was er wie tun soll. Beispiele für Programmiersprachen, die Imperativ sind, sind Assembler, Ada oder Pascal.

Das imperative Konzept ist sehr weit verbreitet, vor allem weil es wesentlich einfacher ist, die Funktionalität eines Programms sowie die Ausführungssequenz nachzuvollziehen, wenn diese explizit gegeben sind. Darüber hinaus hat der Entwickler die komplette Kontrolle über das Verhalten seines Programms. Er kann zum Beispiel Kontrollstrukturen einsetzten, um die Befehlsausführung zu steuern.

3.8 Dynamischer Software-Test

Dynamische Software-Testverfahren sind bestimmte Testmethoden, die mithilfe von Softwaretests, Fehler in der Software aufdecken. So sollen in Abhängigkeit von dynamischen Laufzeitparametern, wie Laufzeitumgebung, auftretende Programmfehler erkannt und lokalisiert werden.

Bei statischen Test-Verfahren wird die zu testende Software nicht ausgeführt. Hingegen muss das Testobjekt im dynamischen Testverfahren mit Eingabedaten versehen und zur Ausführung gebracht werden. Das Grundprinzip der dynamischen Test-Verfahren beruht darauf, festgelegte Testfälle in die zu testende Software systematisch auszuführen. Zusätzlich werden für jeden Testfall die erwarteten Ausgabedaten angegeben. Die vom Testlauf entstandenen Ausgabedaten werden mit den jeweils erwarteten Ausgabedaten verglichen. Ist eine Abweichung vorhanden, so liegt ein Fehler vor. Die Hauptaufgabe der einzelnen Verfahren ist die Bestimmung geeigneter Testfälle für den Software-Test. Eine sinnvolle Vorgehensweise, um gesamte Systeme zu testen, ist eine Testreihe zu bestimmen, die Black-Box-Tests und White-Box-Tests kombiniert [SL11].

3.8.1 White-Box-Verfahren

In White-Box-Testverfahren werden Testfälle auf Basis des Softwarequellcodes bestimmt. Sie werden aufgrund von Wissen über den inneren Aufbau der zu testenden Komponenten entwickelt. Die Testfälle des White-Box-Tests werden direkt aus dem Programm und nicht aus der Spezifikation hergeleitet. Diese Verfahren werden eingesetzt, um die Korrektheit eines Systems zu testen. White-Box-Tests beschränken sich, einzelne Fehler in den jeweiligen Komponenten zu finden. Sie sind nicht in der Lage, das Zusammenspiel mehrerer Komponenten zu testen und mögliche Fehler aufzudecken [SL11].

3.8.2 Black-Box-Verfahren

Black-Box-Testverfahren werden auch funktionsorientierte Testmethoden genannt. Sie überprüfen mit dem Einsatz ausgewählter Testfälle, inwieweit das Testobjekt die

vorgegebenen Spezifikationen erfüllen kann. Besonders interessant ist es, dass diese Verfahren ohne Kenntnisse über den inneren Aufbau des zu testenden Systems angewendet werden. Sie werden allein auf der Basis der Spezifikation entwickelt. Ziel des Black-Box-Testverfahrens ist es, die Übereinstimmung des gesamten Softwaresystems mit der Spezifikation zu überprüfen. Die Testfälle werden direkt aus der Spezifikation abgeleitet. Mit deren Hilfe werden Fehler gegenüber der Spezifikation aufgedeckt. Black-Box-Verfahren betrachten das Testobjekt als Ganzes und sind nicht in der Lage, mögliche Fehler auf Komponentenebene zu identifizieren [SL11].

4 Testkonzept

Ziel dieser Arbeit besteht darin Testfälle zu erstellen, die einen möglichst hohen Überdeckungsgrad der TOSCA-Spezifikation erzielen sollen. Damit sollte überprüft werden, inwieweit die OpenTOSCA (TOSCA-Engine) den TOSCA-Standard erfüllt. Im nächsten Schritt sollen anhand der durchgeführten Tests Fehler oder Probleme in der OpenTOSCA identifiziert und dokumentiert werden.

4.1 Teststrategie

Für das Testen der Kompatibilität und Compliance der OpenTOSCA werden die Blackbox-Verfahren verwendet, da sie bekanntermaßen auf der Spezifikation basieren. Damit wird der innere Aufbau der OpenTOSCA nicht herangezogen und die Testfälle werden direkt aus der TOSCA-Spezifikation abgeleitet. Anschließend werden diese Testfälle in die Engine geladen und ausgeführt. Zum Schluss wird ein Ist-/Sollvergleich vollzogen, um mögliche Fehler festzumachen.
Da allerdings die Testfälle aus mehreren Komponenten bestehen, muss sichergestellt werden, dass diese auch korrekt laufen. Damit kann man gewährleisten, dass der Blackbox-Test korrekte Ergebnisse liefert. Dieses Problem kann gelöst werden, in dem man diese Komponenten einzeln für sich testet. Hierfür werden White-Box-Verfahren eingesetzt [SL11], wie im Abschnitt 3.8.1 bereits beschrieben.

Für diese Arbeit sieht das Testkonzept zwei Testphasen vor:

- In der ersten Testphase werden einzelne Komponenten der Testfälle ermittelt und auf Korrektheit getestet. Als erster Schritt werden die Service Templates aus der TOSCA-Spezifikation abgeleitet und überprüft. Anschließend werden Artefakte, die in den Service Templates gebraucht werden, einzeln getestet. Am Ende dieser Testphase werden Pläne zur Ausführung der Service Templates erstellt und überprüft.

- In der zweiten Testphase werden die Komponenten, die in der ersten Testphase getestet wurden, in Zip-Dateien gepackt. Diese Dateien werden gemäß der TOSCA-Spezifikation jeweils den Namen CSAR tragen. Diese CSARs sind unsere Testfälle, die zusammen mit der TOSCA-Engine (OpenTOSCA) und der Datei mit den erwarteten Ergebnissen (Soll-Werte), den Integrationstest bilden.

Im Folgenden werden die einzelnen Testphasen genauer beschrieben.

4.2 Erste Testphase: Komponententest

Hier werden alle Bausteine der Testfälle einzeln entwickelt, beziehungsweise im Falle der Artefakte ermittelt und getestet. Für die Komponententests werden White-Box-Verfahren eingesetzt. Durch die Vielfalt der zu testenden Komponenten werden verschiedene Entwicklungs- und Testmechanismen zum Einsatz kommen, welche in den folgenden Abschnitten näher beschrieben werden.

4.2.1 Service Template Test

Service Templates werden direkt aus der TOSCA-Spezifikation abgeleitet und bilden den Kern der Testfälle. Deshalb ist eine gründliche Analyse des Nutzens der ausgesuchten Service Templates, sowohl in Bezug auf die Einfachheit als auch in Bezug auf den Überdeckungsgrad der TOSCA-Spezifikation, von sehr hoher Bedeutung. Darauffolgend soll eine Datei erstellt werden, die die vorausgesagten Ergebnisse für die einzelnen Service Templates beinhaltet. Diese Datei wird in der zweiten Testphase als Sollvergleich dienen.

Da die TOSCA-Spezifikation ein XML-Schema ist, werden die daraus abgeleiteten Service Templates in einem XML-Editor erstellt und nach Korrektheit überprüft (siehe Abb.9). Als Werkzeuge kommen ein XML-Editor sowie die TOSCA-Spezifikation zum Einsatz.

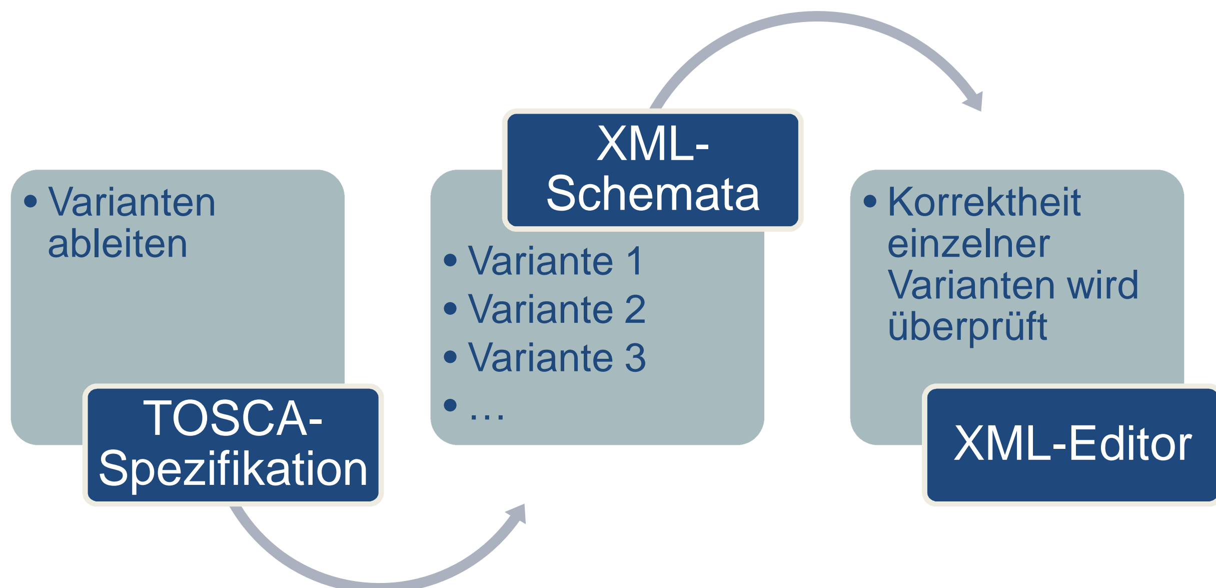

Abbildung 9: Service Templates bzw. Testfälle aus der Spezifikation ableiten

4.2.2 Artefakte Test

Der nächste Baustein der Testfälle sind die Artefakte. Diese ergeben sich direkt aus den bereits entwickelten Service Templates. Mögliche Artefakte können zum Beispiel Images, ausführbare Skripte, JEE Komponenten oder auch Web Services sein.

Diese Artefakte müssen sich jeweils Komponententests unterziehen. Abhängig davon, um welche Art von Artefakten es sich handelt oder in welcher Programmiersprache sie geschrieben sind, werden diese Artefakte unterschiedlich getestet. Beispielsweise wie in Abbildung 10 gezeigt wird, werden für diverse Java-Programme JUnit-Tests eingesetzt. Skriptsprachen müssen Unit-Tests durchlaufen, während die Framework SoapUI für das Testen von Web-Services verwendet werden.

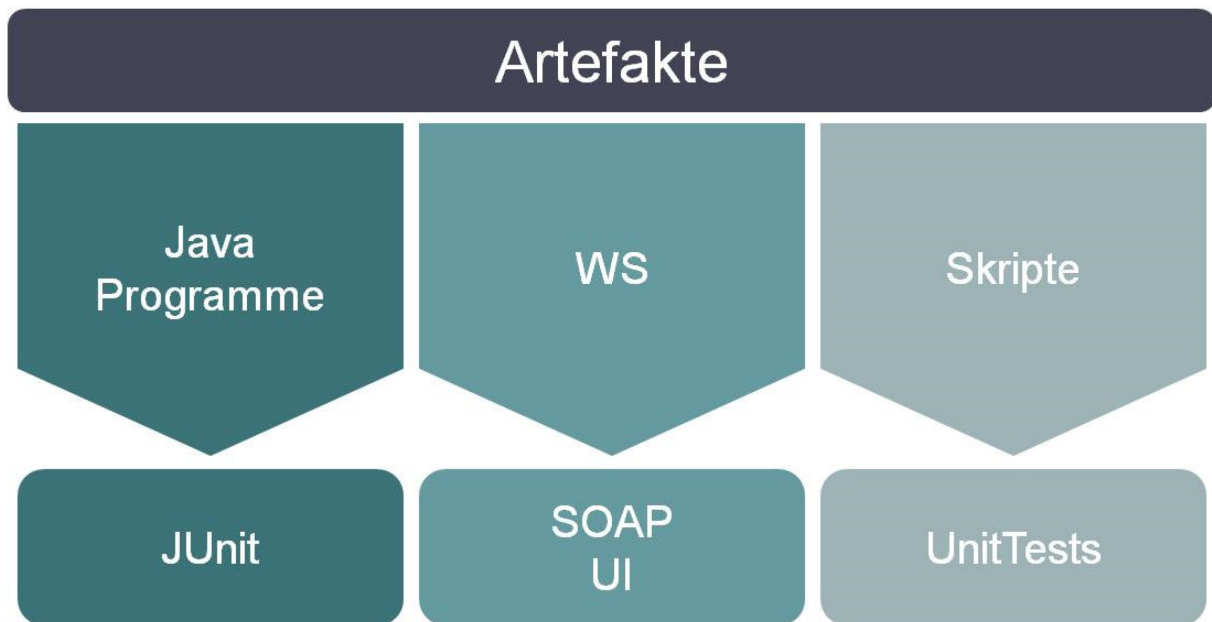

Abbildung 10: Mögliche Artefakte Tests

4.2.3 Plan Test

Der letzte Baustein der Testfälle wäre der Plan. Zur Ausführung der Service Templates, beziehungsweise der Artefakte, werden Workflow-Pläne gebraucht. Daher werden einzelne Pläne für jedes Service Template und die dazugehörigen Artefakte erstellt. Wie in Abbildung 11 zu sehen ist, könnten diese Pläne mit einer Modellierungssoftware wie BPEL-Designer modelliert und getestet werden.

Abbildung 11: Plan Test

4.3 Zweite Testphase: Integrationstest

4.3.1 Testfälle erzeugen

Nachdem alle Komponenten der Testfälle einzeln entwickelt und getestet wurden, wird in dieser Testphase jeweils ein Service Template mit den zugehörigen Artefakten und den zugehörigen Plänen in einem Zip-File namens CSAR gepackt (siehe Abbildung 12). Diese CSAR-Dateien sind die Testfälle, die die Testumgebung dieser Arbeit bilden.

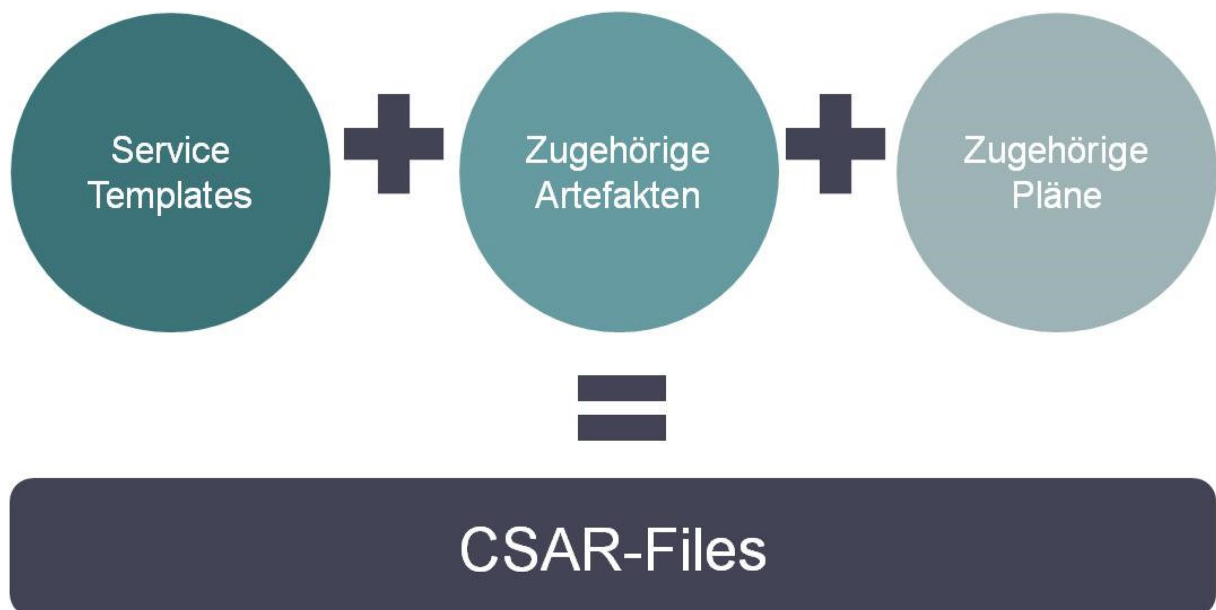

Abbildung 12: CSAR-Dateien (-Files) erzeugen

4.3.2 Integrationstest

Nach dem Abschluss des Komponententests, sowie nach der Erstellung der Testfälle und nach der Bestimmung der vorausgesagten Ergebnissen (Soll-Werte), sind alle Voraussetzungen erfüllt und der Integrationstest kann an dieser Stelle durchgeführt werden.

Wie in Abbildung 13 zu sehen ist, werden die Testfälle (CSARs) einzeln in die OpenTOSCA geladen und ausgeführt. Anschließend folgt ein Ist/Soll-Vergleich. Zum Schluss werden die daraus resultierenden Fehler oder Abweichungen in die Dokumentation eingetragen.

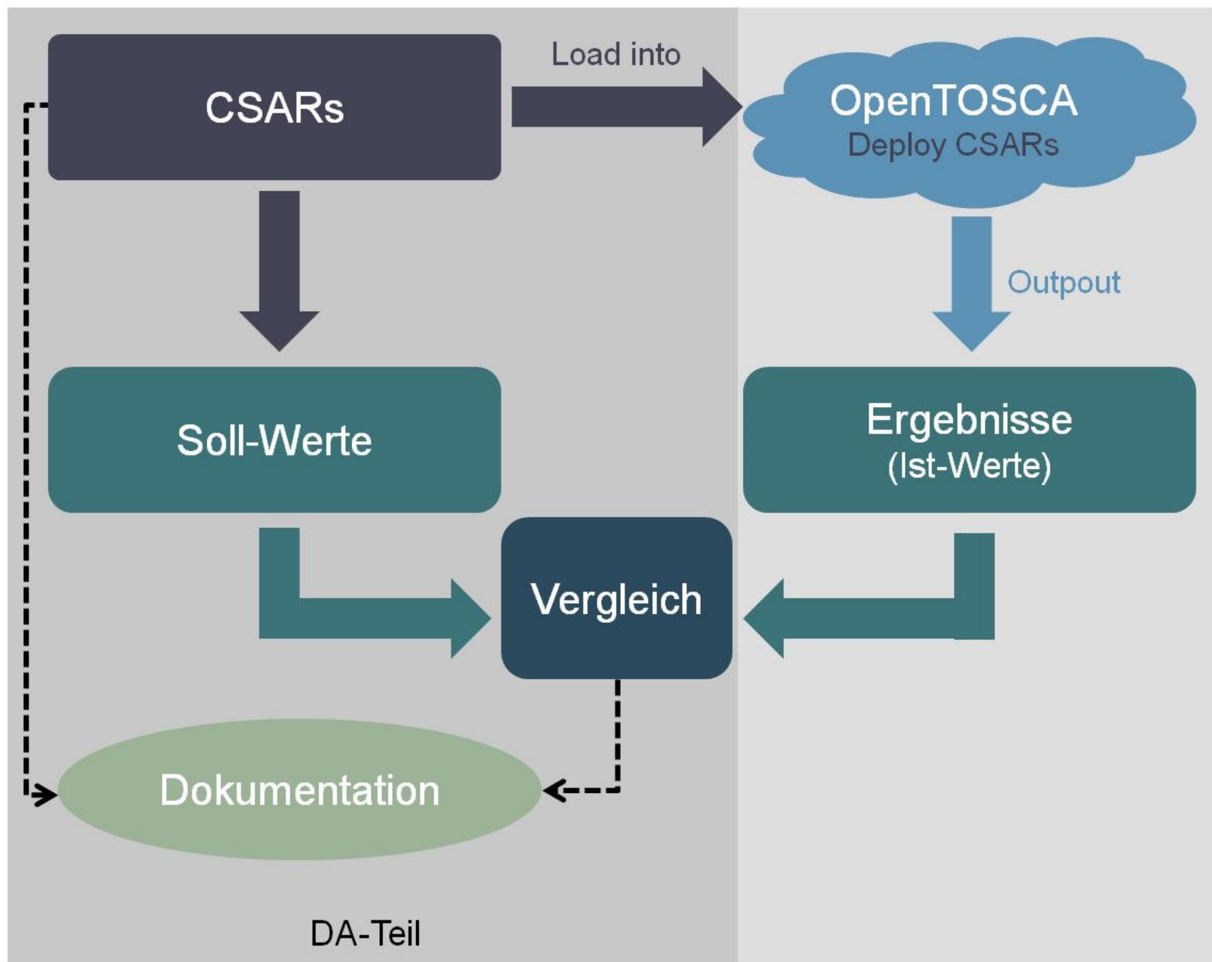

Abbildung 13: Integrationstest

5 Implementierung

In diesem Kapitel wird die Implementierungsphase vorgestellt und anschließend die Ausführung der Testfälle beschrieben. Diese Phase bildet den Kern dieser Diplomarbeit, sie setzt das Testkonzept, welches im vorherigen Kapitel vorgestellt wurde, um.
Zu Beginn wird die erste Testphase des Testkonzepts durchgeführt. In dieser Testphase wird erläutert, wie die einzelnen Komponenten der Testfälle erzeugt und auf Korrektheit getestet werden. Als Nächstes werden die in zwei Gruppen geteilten Testfälle vorgestellt, dabei wird auf die Vorzüge der beiden Testgruppen eingegangen. Mit der Fertigstellung der Testfälle wird darauffolgend die Testumgebung bereitgestellt und die zweite Testphase des Testkonzepts kann dann durchgeführt werden. Nach einer ausführlichen Beschreibung dieser Testphase folgt zum Schluss eine Analyse der Testdurchführung.

5.1 Testfälle Erzeugung

Für die Erstellung der Testfälle werden mehrere Komponenten benötigt. Diese Komponenten stehen in gewisser Beziehung zueinander. In den folgenden Unterpunkten werden sowohl die Entwicklung als auch die Zusammenhänge dieser Komponenten einzeln beschrieben.

Zu diesem Zweck werden verschiedene Werkzeuge zum Einsatz kommen, die dann anhand von Beispielen näher erläutert werden. Allerdings muss an dieser Stelle darauf hingewiesen werden, dass aufgrund der thematischen Ausrichtung dieser Arbeit, auf eine detaillierte Beschreibung dieser Werkzeuge nicht eingegangen werden kann. Für das gründliche Verständnis der Softwares sind die Literaturen der jeweiligen Tools zu empfehlen.

5.1.1 Die TOSCA-Spezifikation

Die TOSCA-Spezifikation dient im Grunde als Grundlage für die Erstellung der Testfälle. Wie bereits unter dem Punkt 3.2 erwähnt wurde, ermöglicht diese Spezifikation durch ein definiertes Meta-Modell die Beschreibung von IT-Diensten. In einem TOSCA-Meta-Modell, auch Service Template genannt, wird die Struktur einer Cloud-Anwendung beschrieben.

Das Service Template bildet wiederum durch ein Topology Template, gemäß der TOSCA-Spezifikation, zusammen mit Plänen und gegebenenfalls anderen Komponenten den Inhalt eines Cloud-Service-Archives (kurz CSAR).

5.1.1.1 Die Service Templates

Die Beschreibung und Entwicklung eines Modells für die Topologie einer Cloud-Anwendung wird in TOSCA anhand von Service Templates erstellt. Wie in Abbildung 14 zu sehen ist, besteht ein Service Template aus einem Topology Template, das sich wiederum aus Node Templates und Relationship Templates zusammensetzt, sowie Types und Pläne. Der gesamte Prozess zur Ausführung und Verwaltung eines Dienstes kann als ein gerichteter Graph, dessen Knoten durch die Node Templates und dessen Kanten durch die Relationship Templates repräsentiert werden, betrachtet werden. Diese Knoten und Kanten bilden zusammen das Topology Template einer Anwendung, das wie eingangs bereits erwähnt einen Teil eines

Service Templates stellt. Dieser Graph (das Topology Template) ist der Modellierungsaspekt in TOSCA. Pläne können auch ein Teil eines Service Templates sein, sie werden in Abschnitt 5.1.3 ausführlich beschrieben [TOSCA13].

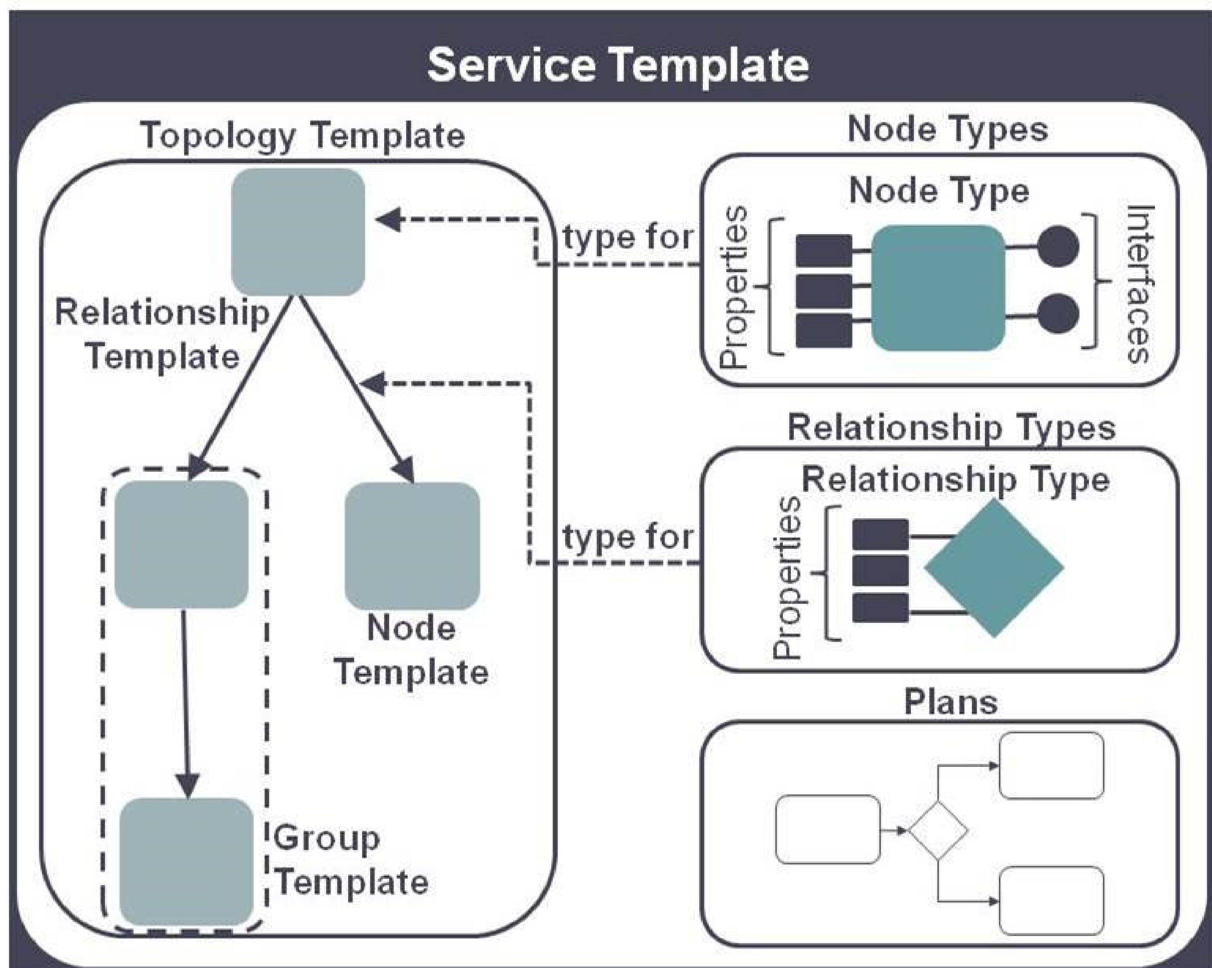

Abbildung 14: Service Template [TOSCA13]

Viel mehr werden wir uns an dieser Stelle auf die Unterschiede bzw. auf die Zusammenhänge zwischen Node bzw. Relationship Templates und Types, sowie auf die Node/ Relationship Type Implementation konzentrieren.

Durch den Einsatz von Node Templates wird festgelegt, wann ein Node Type eintreten kann. Genauer gesagt, ein Node Type kann als Vorlage einer Komponente eines Dienstes verstanden werden und kann damit eine Komponente der Anwendung repräsentieren. Dennoch müssen Eigenschaften und Operationen, die einen Node Type benötigen, um mit der Komponente interagieren zu können, angegeben werden. Hingegen ist das Prinzip eines Node Types, eine abstrakte Beschreibung einer Komponente, die beliebig oft wieder verwendet werden kann, anzubieten.

Ein Node Template nutzt einen Node Type, indem es auf ihn verweist. In anderen Worten, ein Node Template wird durch einen Node Type typisiert. Außerdem kann ein Node Template dem Node Type zusätzliche Informationen geben, wie zum Beispiel welche Parameter genutzt werden müssen oder wie oft die Komponente auftreten soll, um ordnungsgemäß zu funktionieren. Die Implementierung eines referenzierten Node Types wird durch eine Node

Type Implementation dargestellt. Für den Node Type werden darüber hinaus Implementation Artefakte, die die Schnittstellen realisieren, mithilfe der Node Type Implementation definiert. Es können aber auch Deployment Artefakte, die ein Node Template repräsentieren, entweder in einer Node Type Implementation oder auch direkt in dem Node Template definiert werden.

Eine ähnliche Beziehung pflegen Relationship Template und Relationship Type zueinander. Das Relationship Template verweist auf dieselbe Weise auf den letzteren, versorgt ihn mit Informationen über die Richtung einzelner Beziehungen zwischen den Node Templates. Dafür werden Schnittstellen mit Management-Operationen sogenannte „Source Element" und „Target Element" zur Verfügung gestellt, die dann die Richtung der Verbindung zwischen zwei Node Templates definieren. Weitere optionale Einschränkungen können auch im Relationship Template bestimmt werden. Genau wie für Node Types, werden Relationship Type Implementations, die die Implementierung des Relationship Types repräsentieren, eingesetzt.

Zudem können Node Templates gewisse Beziehungen zueinander haben. Diese Beziehungen finden paarweise zwischen zwei Node Templates statt und werden durch Relationship Templates repräsentiert. Damit bilden Node Templates und Relationship Templates zusammen den bereits erwähnten gerichteten Graph und sind somit der Inhalt des Topology Templates. Das Ganze wiederum ist ein Teil eines Service Templates, das alle separaten Teile (das Topology Template, die Node Types, die Relationship Types und einen Plan oder eventuell mehrere Pläne) umfasst [TOSCA13].

In diesem Zusammenhang soll erwähnt werden, dass die TOSCA-Spezifikation es möglich macht, verschachtelte Strukturen zu definieren. Das heißt, dass es durchaus möglich ist, mehrere Service Templates ineinander zu verbinden. Damit kann die Komplexität eines Dienstes beliebig hoch sein.

Es gibt zwei Möglichkeiten ein Service Template zu erzeugen. Die Erste wäre anhand eines XML-Editors. Hierfür wird der XML-Code für das Service Template manuell geschrieben. Zur Veranschaulichung ist ein XML-Code eines Service Templates aus einem Testfall in Listing 1 zu sehen. Die zweite Möglichkeit hingegen, wie in Abbildung 15 gezeigt wird, ist die graphische Modellierung eines Service Templates, die mithilfe von dem Topology-Modeller der Winery-Software erstellt wurde.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Definitions xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
xmlns:ns1="http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes"
xmlns:ns2="http://docs.oasis-open.org/tosca/ns/2011/12/ToscaSpecificTypes"
xmlns:ns3="http://www.example.com/tosca/Types/ExampleWS"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="ExampleWSDefinitions"
name="ExampleWS Definitions" target-
Namespace="http://www.example.com/tosca/ServiceTemplates/ExampleWS"
xsi:schemaLocation="http://docs.oasis-open.org/tosca/ns/2011/12 TOSCA-v1.0-
cs02.xsd">
  <Import namespace="http://example.ws" import-
Type="http://schemas.xmlsoap.org/wsdl/" location="Imports/WS_Example.wsdl" />
  <Import namespace="http://www.example.com/tosca/Types/ExampleWS" import-
Type="http://docs.oasis-open.org/tosca/ns/2011/12" loca-
tion="Definitions/ExampleWSTypes-Definitions.xml" />
  <Import namespace="http://MyTest.com/Test" import-
Type="http://schemas.xmlsoap.org/wsdl/" location="Imports/CallerArtifacts.wsdl" />
  <ServiceTemplate id="ExampleWS" name="ExampleWS Service Template">
    <TopologyTemplate>
```

```
      <NodeTemplate id="ExampleWSTempl" name="ExampleWSService"
type="ns3:ExampleWSNodeType" />
    </TopologyTemplate>
    <Plans target-
Namespace="http://www.example.com/tosca/ServiceTemplates/ExampleWS">
      quupsr
      <Plan id="DeployExampleWS" name="ExampleWS Build Plan" plan-
Type="http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan" planLan-
guage="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
        <PlanModelReference reference="Plans/Caller.zip" />
      </Plan>
    </Plans>
  </ServiceTemplate>
<Definitions >
```

Listing 1: Service Template XML-Code

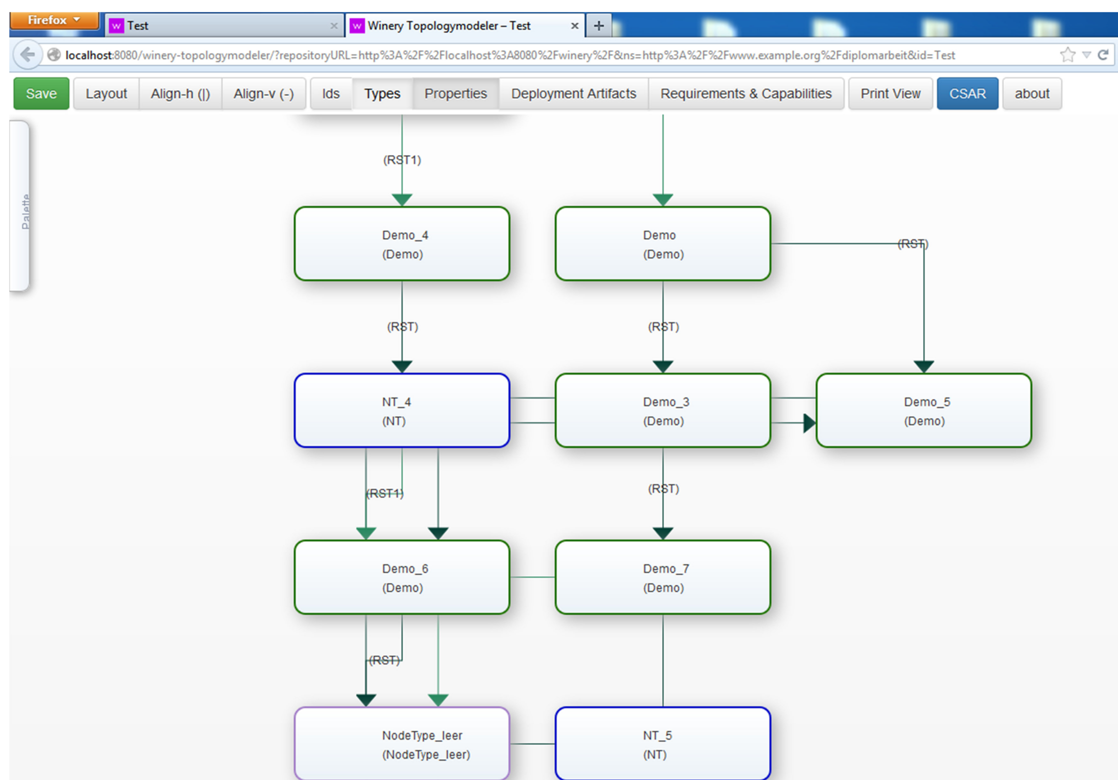

Abbildung 15: Graphische Modellierung eines Service Templates mit Winery

5.1.2 Artefakte

Für die direkte Spezifizierung der Artefakte werden in TOSCA Artifact Templates eingesetzt. Allerdings wäre ein indirekter Zugriff auf ein Artefakt durch ein Artifact Template mithilfe eines URIs, was auf einen Ordner verweist, durchaus möglich. Das Artifact Template muss seinerseits einen vordefinierten Artifact Type, der den Typen des Artefakts vorgibt, nutzen. Der Zugriff auf ein Artefakt kann erfolgen, indem in einer Node Type Implementation oder in einem Node Template ein Verweis auf dessen Artifact Template definiert wird.

TOSCA sieht zwei Arten von Artefakten vor: Implementation Artefakte und Deployment Artefakte.
Der grundsätzliche Unterschied zwischen Implementation Artefakten und Deployment Artefakten hat zum einen mit dem Zeitpunkt, zu dem das Artefakt angewendet wird und zum anderen mit dem Ort, in dem das Artefakt angewendet wird, zu tun.

Implementation Artefakte liefern die Funktionalitäten für den Plan, um die Deployment Artefakte richtig auszuführen. Ein WAR-File ist ein mögliches Beispiel für ein Implementation Artefakt. Hier soll hervorgehoben werden, dass die Implementation Artefakte zuerst ausgeführt werden müssen, bevor die zugehörigen Operationen gestartet werden. Mit anderen Worten, ein TOSCA-Container muss in der Lage sein, die im Testfall benötigten Implementation Artefakte erst laufen zu lassen, um dann die Operationen anwenden zu können. So eine Operation könnte zum Beispiel die Instanziierung von einem Node Type sein. An dieser Stelle ist es ebenfalls wichtig darauf hinzuweisen, dass der TOSCA-Container jedes Implementation Artefakt, das in einem Service Template enthalten und referenziert ist, unterstützten muss. Andernfalls wird eine Fehlermeldung während des Imports auftreten [TOS13].

Damit solch eine Instanziierung erfolgreich durchgeführt werden kann, müssen gewisse Deployment Artefakte (Softwares, Skripte, Image usw.) auf der Zielumgebung zur Verfügung stehen. Zu diesem Zweck unterstützt ein TOSCA-Container eine Reihe von Deployment Artefakt Typen, die er ausführen kann.
Hier gilt allerdings wie für Implementation Artefakte auch, dass ein Service Template, das Deployment Artefakt Typen enthält, die durch den TOSCA-Container nicht gestützt werden, nicht ausgeführt werden kann.

In Abbildung 16 sind die Unterschiede der beiden Artefakt-Gruppen graphisch dargestellt.

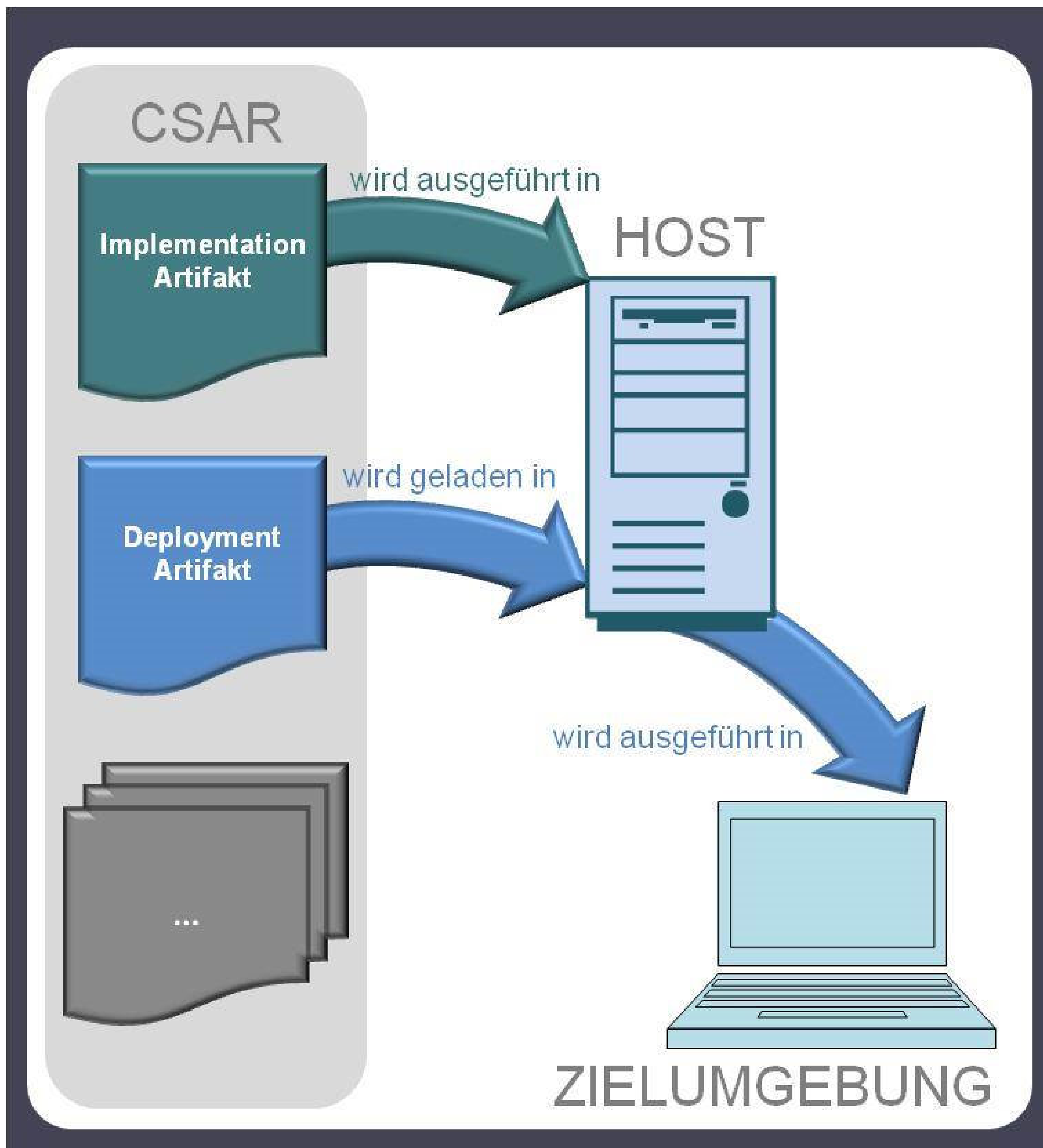

Abbildung 16: Implementation/ Deployment Artefakte

Deployment Artefakte sind wie bereits schon erwähnt fertige Software, die von den jeweiligen Herstellern auf Korrektheit getestet wurden und fehlerfrei funktionieren müssen. Aus diesem Grund sind sie für diese Arbeit relativ uninteressant. Vielmehr wollen wir uns mit den Implementation Artefakten auseinandersetzen. Diese müssen nämlich erst erzeugt werden und im zweiten Schritt auf deren Korrektheit sichergestellt werden.

In diesem Zusammenhang werden wir uns in den kommenden Abschnitten mit der Entwicklung und dem Testen eines Web-Services beschäftigen. Wir werden außerdem sehen, wie dieser Web-Service als WAR-Datei konvertiert werden kann, welche letzten Endes als Implementation Artefakt zur Verfügung stehen wird.

5.1.2.1 Web-Service

Im Allgemeinen werden Web-Services als Grundlage für die Zusammenarbeit von verschiedenen Rechnern, die geographisch verteilt sein können und über ein Netzwerk miteinander verbunden sind, eingesetzt. Informationen über den Aufbau und die Struktur des Web-Services sind von besonderer Wichtigkeit, damit die Kommunikation mit dem Web-Service erfolgreich durchgeführt werden kann. Die Darstellung des Aufbaus der Web-Services kann in Form einer WSDL-Beschreibung erfolgen und festgehalten werden. Durch die Verwendung der WSDL-Beschreibung kann im Grunde festgelegt werden, wie der Zugriff auf einen Web-Service zu erfolgen hat. Daraus kann praktisch eine Art Grundgerüst erstellt werden, was besonders interessant sein kann, vor allem wenn die WSDL-Beschreibung bereits vorhanden ist, denn damit wird der Aufwand für das Programmieren wesentlich vereinfacht.

Das Ziel dieses Abschnittes ist die Erstellung eines Web-Services, der später als WAR-File exportiert wird. Dieses File wird im Folgenden als Implementation Artefakt für einen Testfall zur Verfügung gestellt.

Als Voraussetzung für die erfolgreiche Erstellung und Ausführung eines Web-Services werden folgende Anwendungen benötigt:

- Eine aktuelle Java SE JDK Version muss vorhanden sein.
- Eine Eclipse IDE for Java EE Developers oder Ähnliches muss zur Verfügung stehen.
- Apache Axis Software sollte als Webanwendung auf der Tomcat 7 laufen.
- Apache Tomcat 7.0 muss installiert und in Eclipse eingebunden werden.
- Eventuell sollte die Opensource SoapUI Software zum Testen des Web-Services geladen und installiert werden.

Die bereits aufgezählten Anwendungen sind nicht an bestimmte Plattformen gebunden. Sie sind somit plattformunabhängig und unterstützen die meisten verfügbaren Betriebssysteme.

Zu Beginn wird die Klasse mit der Funktionalität erstellt. Darauffolgend wird aus dieser Klasse die Generierung der WSDL-Datei realisiert. Um es konkreter zu erläutern, es wird in Eclipse das dynamische Web-Projekt erst erfolgreich angelegt, dann kann mit dem Erstellen der Klasse, die die Funktionalität bereitstellt, begonnen werden. Anschließend kann wie eingangs erwähnt die automatische Generierung vom Web-Service durchgeführt werden. Eclipse wird an dieser Stelle ein neues Projekt erzeugen. Mit diesem Projekt kann eine Interaktion zwischen dem Web-Service-Client und dem Web-Service durchgeführt werden. Damit kann unter anderem der Service getestet werden. Außerdem kann die Überwachung der Kommunikation zwischen dem Web-Service und dem Web-Service-Client sichergestellt werden [TU10].
Das folgende Listing 2 zeigt den WSDL-Quellcode von einem Web-Service, der im Rahmen dieser Diplomarbeit für einen Testfall erstellt wurde:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://example.ws"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://example.ws"
xmlns:intf="http://example.ws" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
```

```
  <schema elementFormDefault="qualified" targetNamespace="http://example.ws"
xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="doSomething">
    <complexType>
     <sequence>
      <element name="myinput" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="doSomethingResponse">
    <complexType>
     <sequence>
      <element name="doSomethingReturn" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
  </schema>
 </wsdl:types>

   <wsdl:message name="doSomethingRequest">

      <wsdl:part element="impl:doSomething" name="parameters">

      </wsdl:part>

   </wsdl:message>

   <wsdl:message name="doSomethingResponse">

      <wsdl:part element="impl:doSomethingResponse" name="parameters">

      </wsdl:part>

   </wsdl:message>

   <wsdl:portType name="WS_Example">

      <wsdl:operation name="doSomething">

         <wsdl:input message="impl:doSomethingRequest" name="doSomethingRequest">

       </wsdl:input>

         <wsdl:output message="impl:doSomethingResponse"
name="doSomethingResponse">

       </wsdl:output>

      </wsdl:operation>

   </wsdl:portType>

   <wsdl:binding name="WS_ExampleSoapBinding" type="impl:WS_Example">

       <wsdlsoap:binding style="document"
              transport="http://schemas.xmlsoap.org/soap/http" />

       <wsdl:operation name="doSomething">

              <wsdlsoap:operation soapAction="" />
```

```
            <wsdl:input name="doSomethingRequest">
                  <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="doSomethingResponse">
                  <wsdlsoap:body use="literal" />
            </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="WS_ExampleService">
      <wsdl:port binding="impl:WS_ExampleSoapBinding" name="WS_Example">
         <wsdlsoap:address loca-
tion="http://localhost:8080/Example_WS/services/WS_Example"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Listing 2: WSDL-Quellcode eines Web-Services

In Eclipse gibt es darüber hinaus die Möglichkeit den Web-Service grafisch darzustellen. Die folgende Abbildung 17 illustriert den oben schon gezeigten WSDL-Quellcode des Web-Services grafisch.

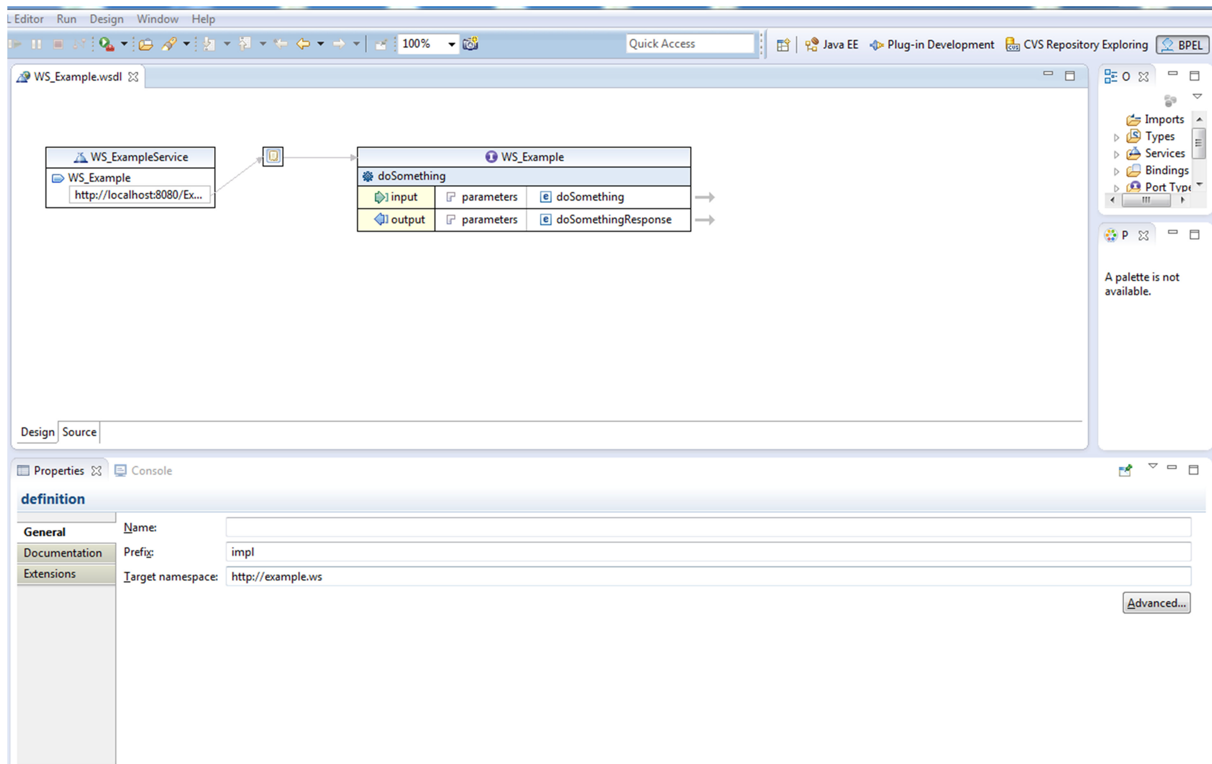

Abbildung 17: WSDL-Graph des Web-Services

Mit dem Web-Services-Explorer könnte man einfach und schnell (wie es in Abb. 18 zu sehen ist) den Web-Service über seine WSDL-Datei testen, sowie SOAP-Request und Response in XML anzeigen lassen. Das Ergebnis der Testdurchführung könnte in der Konsole wie Abbildung 19 zeigt, gelesen werden.

Abbildung 18: Test des Web-Services mit dem Web Services Explorer

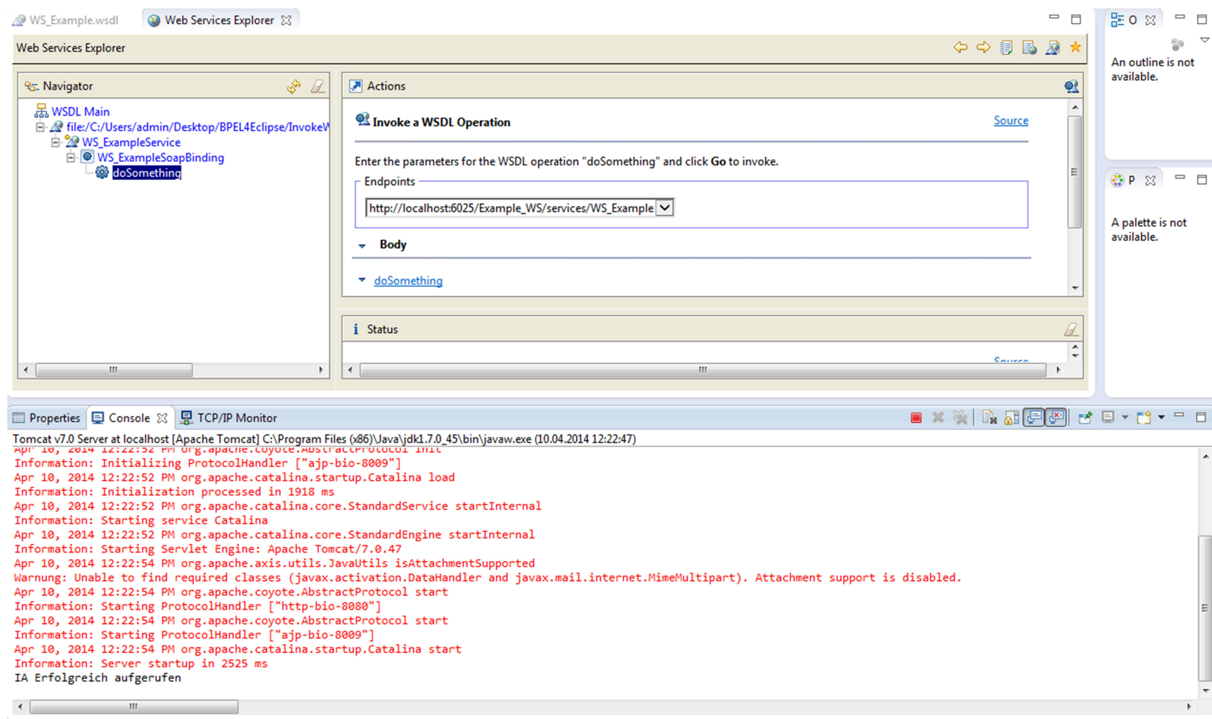

Abbildung 19: Erfolgreiche Ausgabe auf der Konsole

Man kann allerdings andere Softwares zum Testen des Web-Services wie beispielsweise die SoapUI Software, die eine grafische Oberfläche für Webservice-Aufrufe bereitstellt, einsetzen. In Abbildung 20 wird der Web-Service Example_WS mit SoapUI getestet.

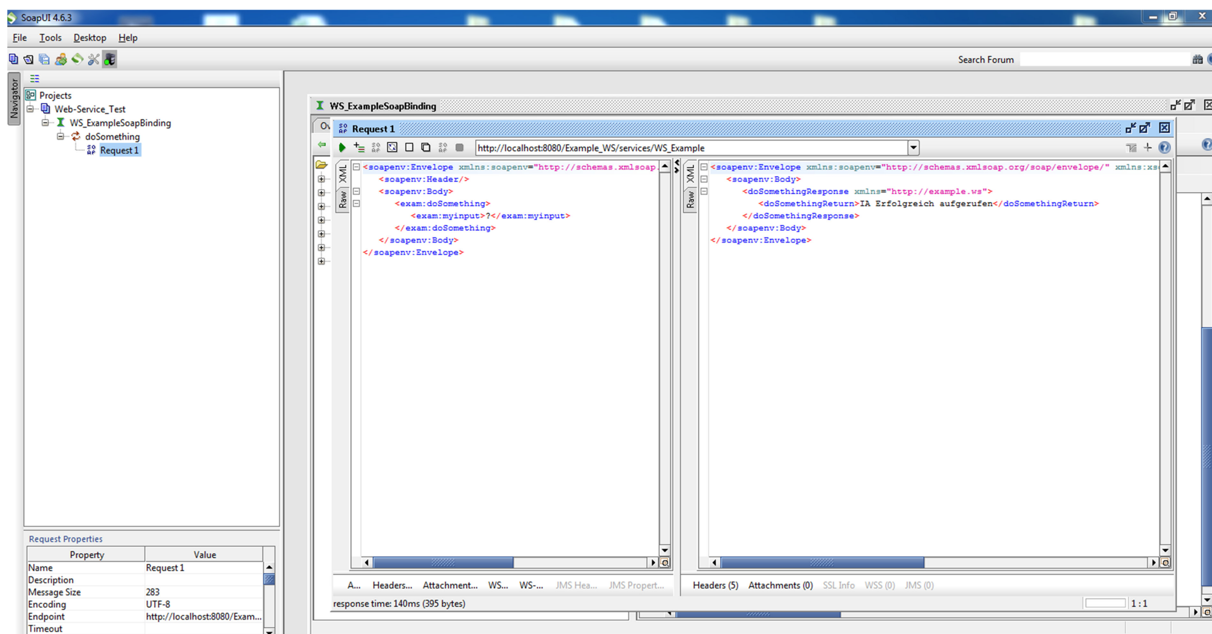

Abbildung 20: Web-Service-Test mit der SoapUI

Nachdem der Web-Service erfolgreich erzeugt und getestet wurde, wird dieser in einem WAR-Archiv verpackt und exportiert. Dies ist in Eclipse mit ein paar Mausklicks leicht und schnell gemacht. Damit wäre der letzte Schritt getan. Der Web-Service steht als WAR-Datei bereit und kann als Implementation Artefakt in einem Testfall (CSAR) genutzt werden.

5.1.3 Pläne

Ein Plan ist ein Workflow, der definieren soll, in welcher Reihenfolge einzelne Operationen ausgeführt werden müssen. Weiter wird in einem TOSCA-Modell durch Pläne das gesamte Management-Verhalten, das einen ganzen Lebenszyklus einer Cloud-Anwendung abdeckt, umgesetzt.

Sowohl die Node Types und Templates als auch die Relationship Types und Templates beschreiben, wie in den vorherigen Abschnitten bereits dargelegt wurde, eine generische Struktur eines Cloud-Dienstes und beinhalten vor allem alle notwendige Informationen über einen Cloud-Dienst. Durch das Ausführen einer Reihe von Befehlen soll der Plan wiederum dazu dienen, diese Daten zu interpretieren, um eine lauffähige Instanz zu erzeugen. Damit entsteht ein konkreter Service, der durch das Service Template repräsentiert wird. In TOSCA wird die Verwendung vom BPEL-Standard für die Erstellung der Pläne erlaubt.

Allerdings kann sich eine vollständige Beschreibung eines BPEL Plans in XML über mehrere Bildschirmseiten erstrecken, was die Übersicht erschweren könnte (ein Ausschnitt vom Quellcode eines BPEL-Plans, der die im vorherigen Abschnitt erstellte WAR-Datei verwaltet, ist in Listing 3 zu sehen). Aufgrund ihrer einfachen Bedienbarkeit und der großen Übersicht bevorzugen Entwickler den Einsatz graphischer Modellierungstools für die Erstellung der BPEL-Prozesse. Diese Modellierungstools bieten unter anderem eine graphische Darstellung der BPEL-Pläne. Sie unterstützen außerdem mit ihrer Funktionalität die Validierung und Codegenerierung [BPE07].

```
<!-- Caller BPEL Process [Generated by the Eclipse BPEL Designer] -->
<!-- Date: Thu Feb 06 14:47:56 CET 2014 -->
<bpel:process name="Caller"
         targetNamespace="http://MyTest.com/Test"
         suppressJoinFailure="yes"
         xmlns:tns="http://MyTest.com/Test"
         xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:ns1="http://example.ws">

    <!-- Import the client WSDL -->
    <bpel:import namespace="http://example.ws" location="WS_Example.wsdl" import-
Type="http://schemas.xmlsoap.org/wsdl/"></bpel:import>
    <bpel:import                                   location="CallerArtifacts.wsdl"
namespace="http://MyTest.com/Test"
              importType="http://schemas.xmlsoap.org/wsdl/" />

    <!-- ================================================================= -->
    <!-- PARTNERLINKS                                                      -->
    <!-- List of services participating in this BPEL process               -->
    <!-- ================================================================= -->
    <bpel:partnerLinks>
        <!-- The 'client' role represents the requester of this service. -->
        <bpel:partnerLink name="client"
                    partnerLinkType="tns:Caller"
```

```
                    myRole="CallerProvider"
                    />
        <bpel:partnerLink name="DSLink" partnerLinkType="tns:DSLinkType" partner-
Role="DSProvider"></bpel:partnerLink>
    </bpel:partnerLinks>

    <!-- ================================================================= -->
    <!-- VARIABLES                                                         -->
    <!-- List of messages and XML documents used within this BPEL process  -->
    <!-- ================================================================= -->
    <bpel:variables>
        <!-- Reference to the message passed as input during initiation -->
        <bpel:variable name="input"
                  messageType="tns:CallerRequestMessage"/>

        <!--
          Reference to the message that will be returned to the requester
          -->
        <bpel:variable name="output"
                  messageType="tns:CallerResponseMessage"/>
        <bpel:variable                  name="DSLinkRequest"                  mes-
sageType="ns1:doSomethingRequest"></bpel:variable>
        <bpel:variable                 name="DSLinkResponse"                  mes-
sageType="ns1:doSomethingResponse"></bpel:variable>
    </bpel:variables>

    <!-- ================================================================= -->
    <!-- ORCHESTRATION LOGIC                                               -->
    <!-- Set of activities coordinating the flow of messages across the    -->
    <!-- services integrated within this business process                  -->
    <!-- ================================================================= -->
    <bpel:sequence name="main">

        <!-- Receive input from requester.
             Note: This maps to operation defined in Caller.wsdl
             -->
        <bpel:receive name="receiveInput" partnerLink="client"
                 portType="tns:Caller"
                 operation="process" variable="input"
                 createInstance="yes"/>

        <!-- Generate reply to synchronous request -->
        <bpel:assign validate="no" name="Assign">
            <bpel:copy>
                <bpel:from>
                    <bpel:literal            xml:space="preserve"><xsd:doSomething
xmlns:xsd="http://example.ws">
                            <xsd:myinput></xsd:myinput>
                        </xsd:doSomething></bpel:literal>
                </bpel:from>
                <bpel:to part="parameters" variable="DSLinkRequest"></bpel:to>
            </bpel:copy>
            <bpel:copy>
                <bpel:from part="payload" variable="input">
                    <bpel:query                                          queryLan-
guage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                        <![CDATA[tns:input]]>
                    </bpel:query>
                </bpel:from>
                <bpel:to part="parameters" variable="DSLinkRequest">
```

```
                    <bpel:query                                                queryLan-
guage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"><![CDATA[ns1:myinput]]></bp
el:query>
                </bpel:to>
            </bpel:copy>
        </bpel:assign>
        <bpel:invoke  name="Invoke"  partnerLink="DSLink"  operation="doSomething"
portType="ns1:WS_Example"        inputVariable="DSLinkRequest"        outputVaria-
ble="DSLinkResponse"></bpel:invoke>
        <bpel:assign validate="no" name="Assign1">
            <bpel:copy>
                <bpel:from>
                    <bpel:literal         xml:space="preserve"><tns:CallerResponse
xmlns:tns="http://MyTest.com/Test">
                            <tns:result></tns:result>
                        </tns:CallerResponse></bpel:literal>
                </bpel:from>
                <bpel:to part="payload" variable="output"></bpel:to>
            </bpel:copy>
            <bpel:copy>
                <bpel:from part="parameters" variable="DSLinkResponse">
                    <bpel:query                                                queryLan-
guage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                        <![CDATA[ns1:doSomethingReturn]]>
                    </bpel:query>
                </bpel:from>
                <bpel:to part="payload" variable="output">
                    <bpel:query                                                queryLan-
guage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"><![CDATA[tns:result]]></bpe
l:query>
                </bpel:to>
            </bpel:copy>
        </bpel:assign>
        <bpel:reply name="replyOutput"
               partnerLink="client"
               portType="tns:Caller"
               operation="process"
               variable="output"
               />
    </bpel:sequence>
</bpel:process>
```

Listing 3: XML-Code Ausschnitt aus einem Testfall in BPEL-Designer

5.1.3.1 Modellierungstool

Für diese Arbeit wurde für die Erstellung der BPEL-Pläne der Eclipse BPEL-Designer als Modellierungswerkzeug eingesetzt.

Der BPEL-Designer ist ein auf der Opensource-Umgebung Graphical Editing Framework (kurz: GEF) basierter Editor. Dieses Modellierungswerkzeug wird zur Erstellung sowie auch zur Bearbeitung von graphischen BPEL-Prozessen verwendet. Besonders für diese Arbeit werden damit die Pläne für die Testfälle modelliert. Als Modell nutzt der BPEL-Designer die Eclipse Modeling Framework (kurz: EMF), die die WS-BPEL 2.0 Spezifikation repräsentiert und die automatisierte Erzeugung von Quellcodes aus den vorhandenen Modellen ermöglicht. Der Designer besitzt die Komponente Validator, die ihre Funktion auf dem EMF-Modell

findet. Diese Komponente ist für die Erstellung von Fehlermeldungen und Warnungen gemäß der Spezifikation zuständig. Für das Deployment und die Ausführung der BPEL-Prozesse in die BPEL-Engine nutzt der BPEL-Designer eine erweiterbare Komponente: die Runtime Framework [SD08].

Die folgende Abbildung 21 stellt die graphische Modellierung von den in Listing 3 vorgestellten Quellcodes mit dem Eclipse BPEL-Designer als BPEL-Plan dar:

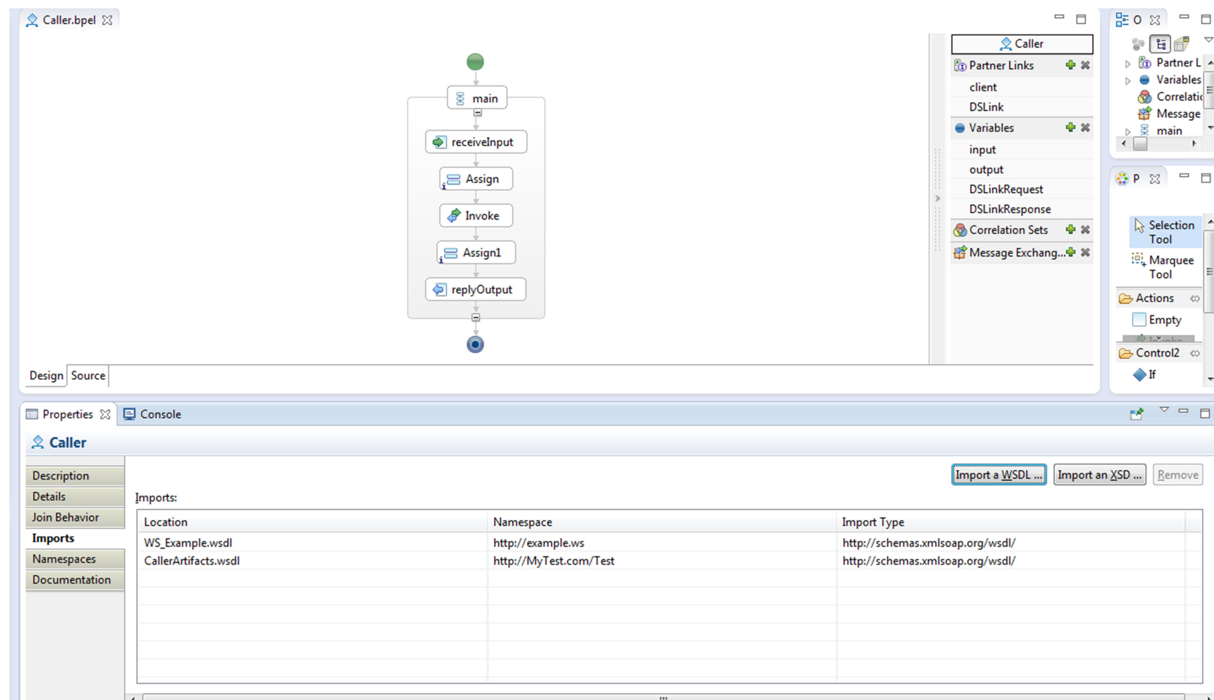

Abbildung 21: Graphische Modellierung des XML-Codes von Listing 3 mit BPEL-Designer

5.1.3.2 WSO2 BPS

WSO2 Business-Prozess-Server (BPS) ist ein benutzerfreundlicher Business Prozess Server, der Geschäfts-Prozesse, die nach dem WS-BPEL geschrieben wurden, ausführt. Er ist ein Opensource-Produkt und ist unter der Apache-Software-Lizenz verfügbar. WS-BPEL ist der Standard, um mehrere synchrone und asynchrone Web-Services in kollaborative und flüssige Prozesse umzusetzen. Darüber hinaus bietet WSO2 BPS eine komplette web-basierte graphische Oberfläche, um Business-Prozesse bzw. Instanzen auszuführen, zu steuern und zu überwachen. Des Weiteren werden Business-Prozesse mithilfe von WS-Security in WSO2 BPS unterstützt. So wird beispielsweise durch diese Unterstützung das Aufrufen von Partner-Services besser geschützt. Neben den bereits erwähnten Funktionalitäten, bietet WSO2 weitere Funktionen wie zum Beispiel das Transportmanagement, die System-Überwachung, die Try-it-Funktion für Business-Prozesse oder auch die SOAP-Nachricht-Verfolgung. Mit WSO2 BPS ist es durch ein Reinigungs-Tool sogar möglich, alte BPEL-Prozess-Versionen zu entfernen. Abbildung 22 zeigt einen mit BPEL-Designer samt Web-Service erstellten Plan, der in WSO2 BPS geladen wurde. Der Plan ist sowohl grafisch als auch in XML-Code zu sehen. Zudem stehen diverse weitere Funktionen zur Verfügung, wie es im linken Teil der Abbildung 22 unten zu sehen ist [WSO13].

Abbildung 22: WSO2 BPS

Für die Diplomarbeit wird WSO2 BPS zum Ausführen der in BPEL-Designer modellierten Pläne eingesetzt. Wie oben bereits erwähnt und veranschaulicht wurde, wird mithilfe der Try-it-Funktion (siehe Abb. 23) die Korrektheit der Pläne, die den Web-Service ausführen und verwalten sollen, überprüft und sichergestellt. Der Web-Service des Testfalls sollte „IA erfolgreich aufgerufen“ im rechten Ausgabefenster der Abbildung als Ausgabe liefern. Wenn das nicht der Fall ist, so wäre ein Fehler vorhanden. Dies muss erst behoben werden und der Test muss anschließend wiederholt werden, bis das richtige Ergebnis geliefert wird.

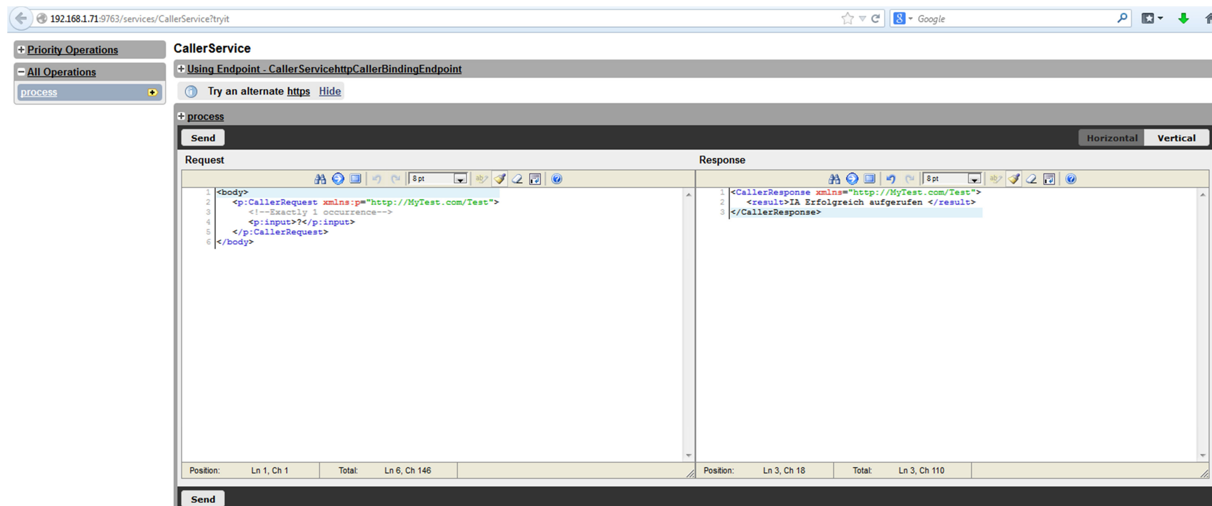

Abbildung 23: Try-It-Funktion des WSO2 BPS

5.1.4 Erstellung der CSARs

Um Unklarheiten vorzubeugen, ist es zunächst notwendig, die Bedeutung vom Begriff „Testfall“ in dieser Arbeit klarzustellen. Im Rahmen dieser Arbeit, wenn es die Rede von einem Testfall ist, wird damit automatisch ein Cloud-Service-Archive (kurz CSAR) gemeint und umgekehrt genauso. Anders formuliert sind die Testfälle dieser Arbeit CSAR-Dateien, das heißt, eine CSAR-Datei repräsentiert einen Testfall.

Bevor es mit der Entwicklung der CSARs gestartet wird, muss das Gesamtdesign definiert werden. Sobald eine genaue Vorstellung davon vorliegt, was ausgeführt werden sollte, kann die tatsächliche Implementierung beginnen.
Der TOSCA-Container-File (kurz CSAR) muss mindestens ein Service Template beinhalten, um die Topologie zu definieren. Auch mögliche Artefakte und Softwares sowie alle Node Templates, Relationship Templates, Pläne und alle weiteren notwendigen Informationen werden in CSAR unter den vorgegebenen Ordnern gespeichert. Anschließend wird diese Ordnerstruktur in einem Archiv (Zip-File) gepackt. Die Endung des Filenamens wird dann mit „.csar“ umbenannt [TOSCA13].

Es gibt zwei Möglichkeiten die CSARs zu erstellen. Die Erste ist die manuelle Möglichkeit (siehe Abbildung 24(a)). Hier wird das Service Template mithilfe eines Editors in XML manuell programmiert. Die entsprechenden Artefakte, Importe und Pläne, die für den jeweiligen Testfall gebraucht werden, werden zusammen wie in den vorherigen Abschnitten erläutert wurde, manuell erstellt und gemäß der Ordnerstruktur gespeichert. Danach wird das Ganze, wie bereits erwähnt, in einem Archiv gepackt.
Die zweite Möglichkeit besteht darin, mithilfe der grafischen Plattform Winery (Abbildung 24(b)) CSAR-Dateien generieren zu lassen, indem man die gesamte Topologie in einem Service Template grafisch erstellt. Dies bietet den Vorteil, dass der Programmieraufwand in XML erspart werden kann. Dennoch müssen die benötigten Artefakte, die bereits wie beispielsweise unter dem Punkt 5.1.2 als WAR-Datei erstellt wurden, sowie auch die dazu erzeugten BPEL-Pläne, manuell in die Winery hochgeladen und eingefügt werden. Winery bietet die Funktionalität das fertige Service Template, samt Artefakten und Plänen, als vollständige CSAR-Datei zu importieren. Beide Varianten sind zielführend und wurden erfolgreich eingesetzt.

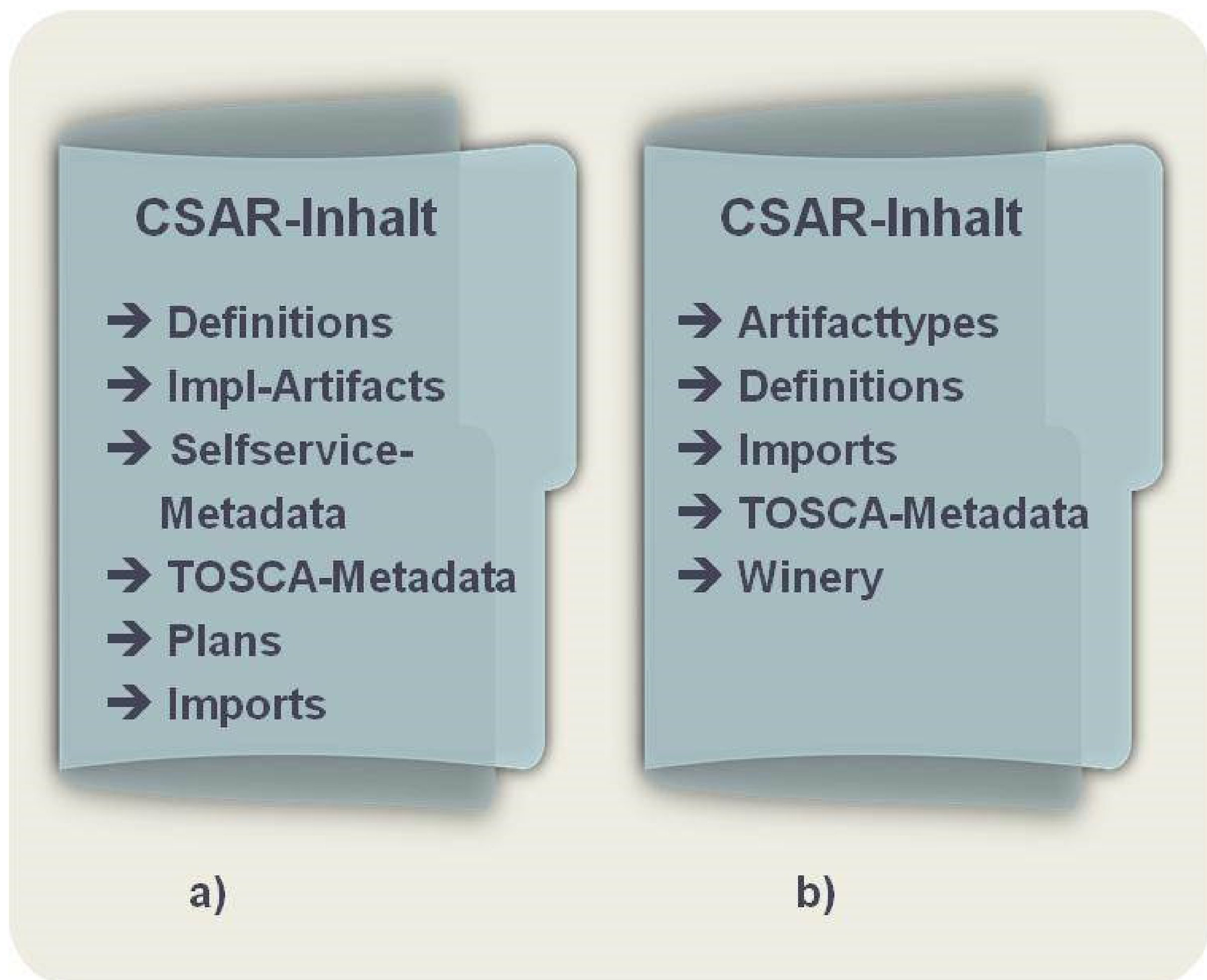

Abbildung 24: CSAR-Ordnerstrukur manuell (a) und mit Winery (b) erzeugt

Als Nächstes wurden die Testfälle dieser Arbeit in zwei Gruppen geteilt. Die erste Testgruppe trägt den Namen „Installable Testfälle“, während die zweite Testgruppe „Deployment Testfälle“ heißt. Die Idee hinter der ersten Testgruppe ist, eine Gruppe von Testfällen zu erzeugen, die möglichst viele Varianten, welche gemäß dem TOSCA-Standard realisierbar sind, abdecken. Diese müssen vom Testobjekt geladen und installiert werden. Daher die Namensgebung „Installable“. Achtzehn CSAR-Dateien bilden den Inhalt dieser Testgruppe. Die zweite Testgruppe „Deployment Testfälle“ beinhaltet CSAR-Dateien, die unter anderem Artefakte und Pläne haben. Damit werden echte Dienste repräsentiert. Anders als bei den „Installable Testfälle“ werden mit diesen CSAR-Dateien echte Anwendungen simuliert. Hiermit sollte geprüft werden, inwieweit das Testobjekt einen gemäß der TOSCA-Spezifikation erstellten und realitätsnahen Service laden und ausführen kann.

Bei der ersten Gruppe „Installable Testfälle“ ist als Ergebnis zu erwarten, dass diese Testfälle durch das Testobjekt, in diesem Fall die OpenTOSCA Engine, mithilfe dessen Benutzeroberfläche, geladen und installiert werden. Darüber hinaus müssen diese Testfälle in dem Container API der OpenTOSCA automatisch installiert und gespeichert werden. Außerdem müssen erfolgreiche Meldungen in den jeweiligen Logfile-Dateien des OpenTOSCAs und des Tomcats zu lesen sein. Mögliche Fehler oder Abweichungen, die entstehen könnten, werden gegebenenfalls ausführlich beschrieben und dokumentiert.

Hingegen, für die zweite Gruppe „Deployment Testfälle“ muss neben den bereits für die erste Gruppe zu erwarteten Ergebnissen zusätzlich gelten, dass die Pläne sowie auch die Artefakte erfolgreich geladen und installiert werden. Die Überprüfung muss dann dazu in dem jeweiligen Business Prozess Server WSO2 BPS für die Pläne und in dem Web-Applikation-Server Tomcat7 für die Artefakte durchgeführt werden.

5.2 Die Testumgebung bereitstellen

Bevor die zweite Testphase gestartet werden kann, muss die Testumgebung bereitgestellt werden. Hierfür kommen neben den bereits eingesetzten Werkzeugen für die Erstellung der Testfälle, sowie neben den erzeugten Testfällen selbst, auch andere Komponenten zum Einsatz. Diese sollen die Rahmenbedingungen für eine erfolgreiche Testdurchführung der Tests ermöglichen.

Die Testumgebung besteht für diese Arbeit aus einem Testrahmen und den benötigten Werkzeugen für die Durchführung der Tests. Der Testrahmen beinhaltet das Testobjekt, sowie eine für das Testobjekt definierte Systemumgebung. Im Rahmen dieser Arbeit wird die OpenTOSCA Engine, die bereits unter dem Punkt 3.3 ausführlich vorgestellt wurde, das Testobjekt sein. Folgende Voraussetzungen müssen für die Systemumgebung gegeben sein, damit eine Testdurchführung erfolgen kann:

- Ein Rechner mit einer starken Kapazität sollte vorhanden sein, um die Ausführung in einer durchaus annehmbaren Zeit durchführen zu können.
- Eine Internetverbindung ist nicht notwendig, wenn alle Softwares auf dem Rechner lokal installiert sind.
- Als Betriebssystem kann zum Beispiel das Betriebssystem Windows 7 oder auch ein Linux System eingesetzt werden.
- Ein Web-Applikation-Server Apache Tomcat muss runtergeladen und installiert werden. Hier sollte darauf geachtet werden die Version 7 zu verwenden.
- Ein Business-Prozess-Server WSO2 BPS muss ebenfalls zur Verfügung stehen.

In Abbildung 25 wird die ganze Testumgebung dieser Arbeit grafisch dargestellt.

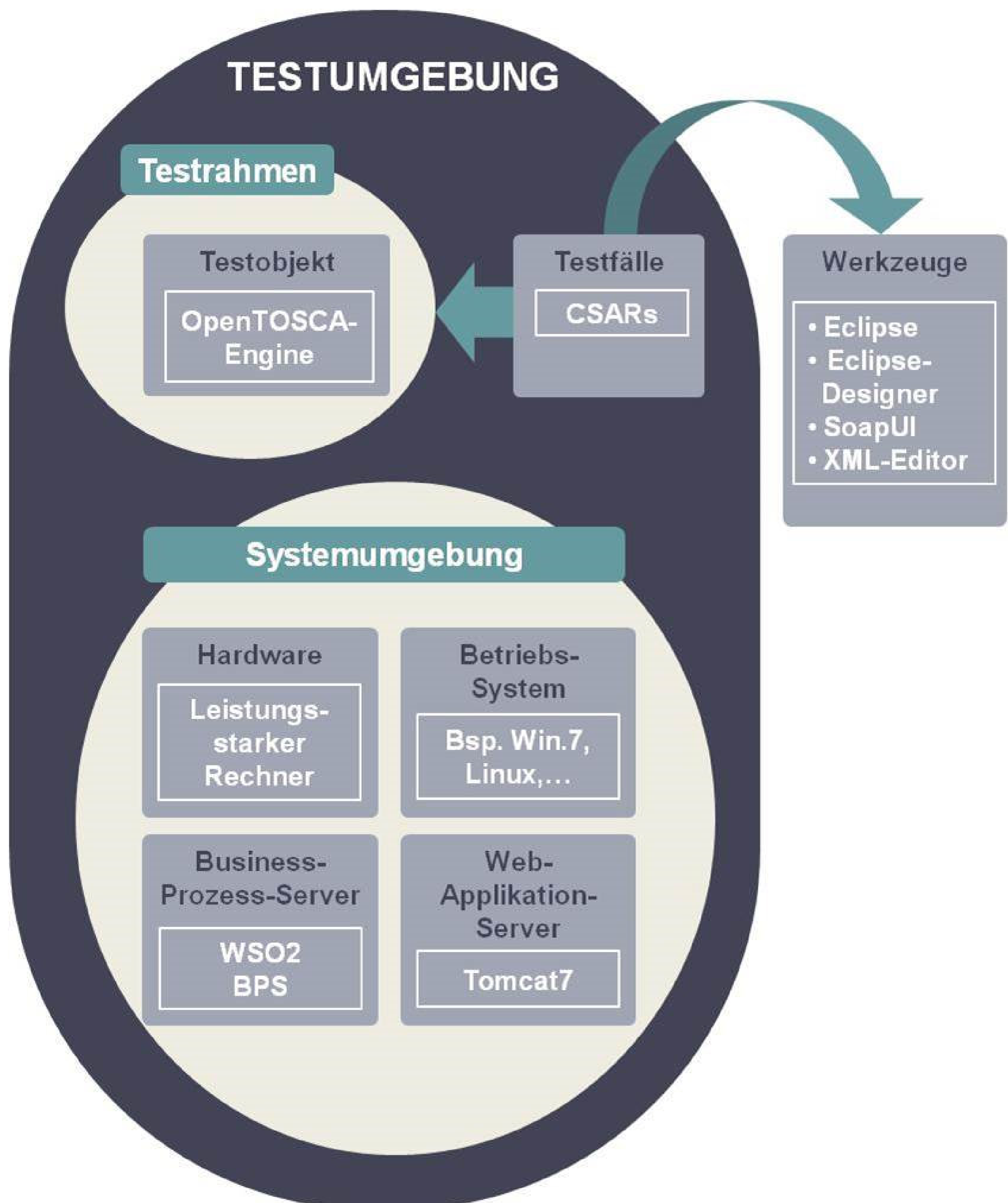

Abbildung 25: Die Testumgebung

5.3 Ausführung der Testfälle

Die Ausführung der ersten Testgruppe „Installable Testfälle“ wird, nachdem alle für die erfolgreiche Testdurchführung notwendigen Voraussetzungen erfüllt wurden, gestartet. Zur Erinnerung, diese Gruppe von Testfällen beinhaltet einige unterschiedliche CSARs, die die TOSCA-Spezifikation ziemlich hoch abdecken und darauf abzielen, die OpenTOSCA-Engine auf ihrer Kompatibilität und Compliance gemäß dem TOSCA-Standard zu testen. Konkreter gesagt, muss die OpenTOSCA in der Lage sein, diese Testfälle zu laden und zu speichern.

Die Verarbeitung eines CSARs in OpenTOSCA beginnt mit dem Aufruf vom Inhalt des Ordners „Definitions“ durch die Komponente TOSCA-Engine. Hier werden unter anderem mögliche Importe bzw. Referenzen aufgelöst. Dieser Schritt spielt zwar für die erste Testgruppe keine Rolle, da hier keine Importe gebraucht werden, jedoch tritt dieser Fall bei der Ausführung von CSARs der zweiten Testgruppe auf. Die Daten, die während der Nutzung von OpenTOSCA erzeugt wurden, werden auf einer lokalen Datenbank gespeichert. Die CSARs der ersten Testgruppe wurden direkt aus der Spezifikation abgeleitet. Sie repräsentieren keine echte Anwendung, hingegen sollen sie viel mehr eine große Anzahl unterschiedlicher Topologien, die von der TOSCA-Spezifikation erlaubt sind, simulieren. Damit wird einen hohen Abdeckungsgrad der Spezifikation erzielt, was die Mindestanforderung an einer TOSCA-Laufzeitumgebung sein sollte. Zum Beispiel sind in den CSARs Service Templates zu finden, die aus einem einzigen Node Template bestehen, zwei oder mehrere Node Templates beinhalten, diese wiederum sind paarweise oder alle miteinander verbunden. Auch zusätzliche Eigenschaften können diese Node Templates besitzen.

Weiter wurden die CSARs dieser Testgruppe einzeln durch die OpenTOSCA Engine erfolgreich geladen und installiert. Diese sind nämlich durch die grafische Benutzerschnittstelle der OpenTOSCA, wie es in den Abbildungen 26, 27 und 28 zu sehen ist, mit Erfolg geladen. Sie wurden aber auch im Container API, wie Abbildung 29 deutlich zeigt, gespeichert. Außerdem wurde die erfolgreiche Ausführung anhand der Meldungen in den Logfiles der OpenTOSCA und der Tomcat bestätigt. Dennoch ist festzustellen, dass auf der Benutzeroberfläche die Meldung „CSAR Status: an error occured during CSAR deployment“ erschienen ist. Diese Meldung wurde mehrmals überprüft, jedoch wurden es weder Fehler in den Logfiles noch im Container identifiziert. Im zweiten Teil der Testdurchführung (die „Deployment Testfälle“) kommen diese Fehlermeldungen nicht vor. Offensichtlich meldet OpenTOSCA immer diesen Fehler, wenn kein Plan in dem CSAR enthalten ist, was mit dem TOSCA-Standard nicht ganz übereinstimmt. Denn, gemäß der Spezifikation, ist es durchaus möglich CSARs auch ohne Pläne zu erstellen.

Auf den kommenden Seiten folgen vier Abbildungen, die das erfolgreiche Laden und Installieren der Testfälle 1 und 18 veranschaulichen. Diese Testfälle sind stellvertretend für alle anderen Testfälle dieser Gruppe, weil die Ergebnisse der Ausführung für alle anderen Testfälle dieser Gruppe mit denen, die in den Abbildungen unten zu sehen sind, übereinstimmten. Aufgrund der Übersichtlichkeit wird bewusst auf die anderen Abbildungen verzichtet.

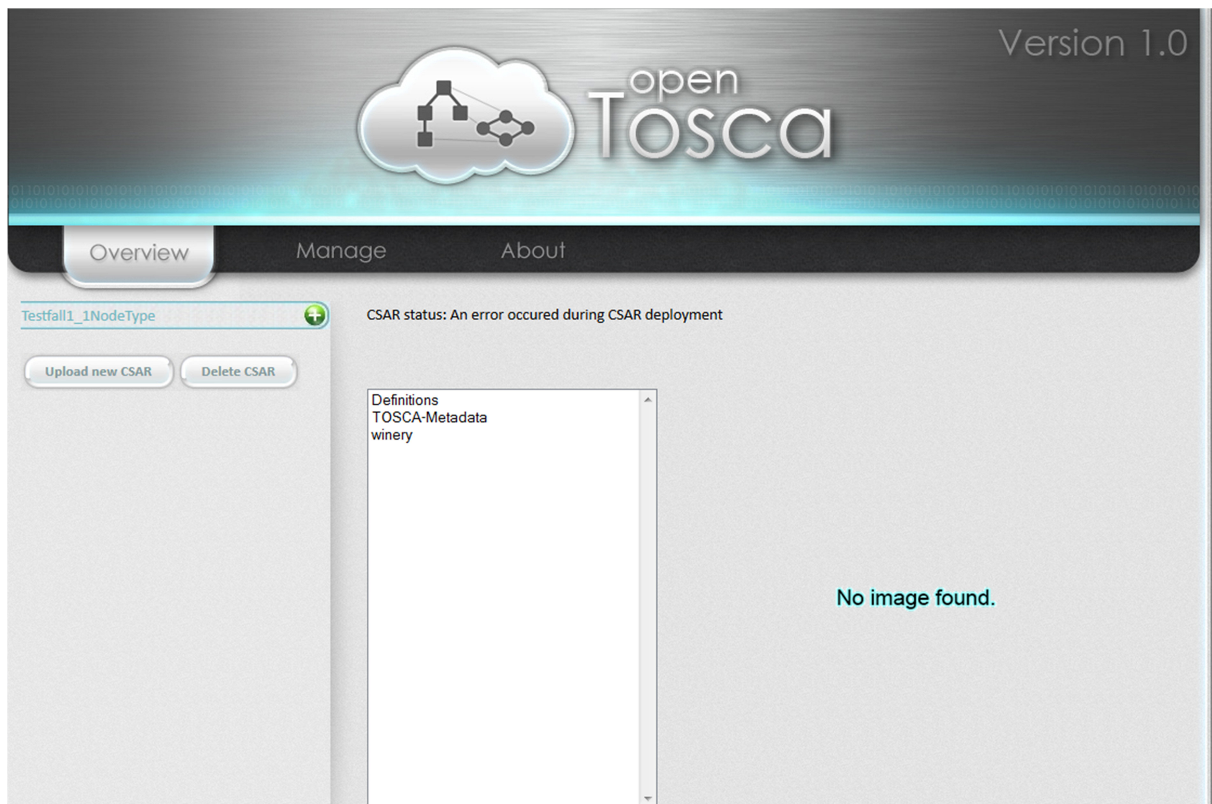

Abbildung 26: Testfall 1 erfolgreich geladen

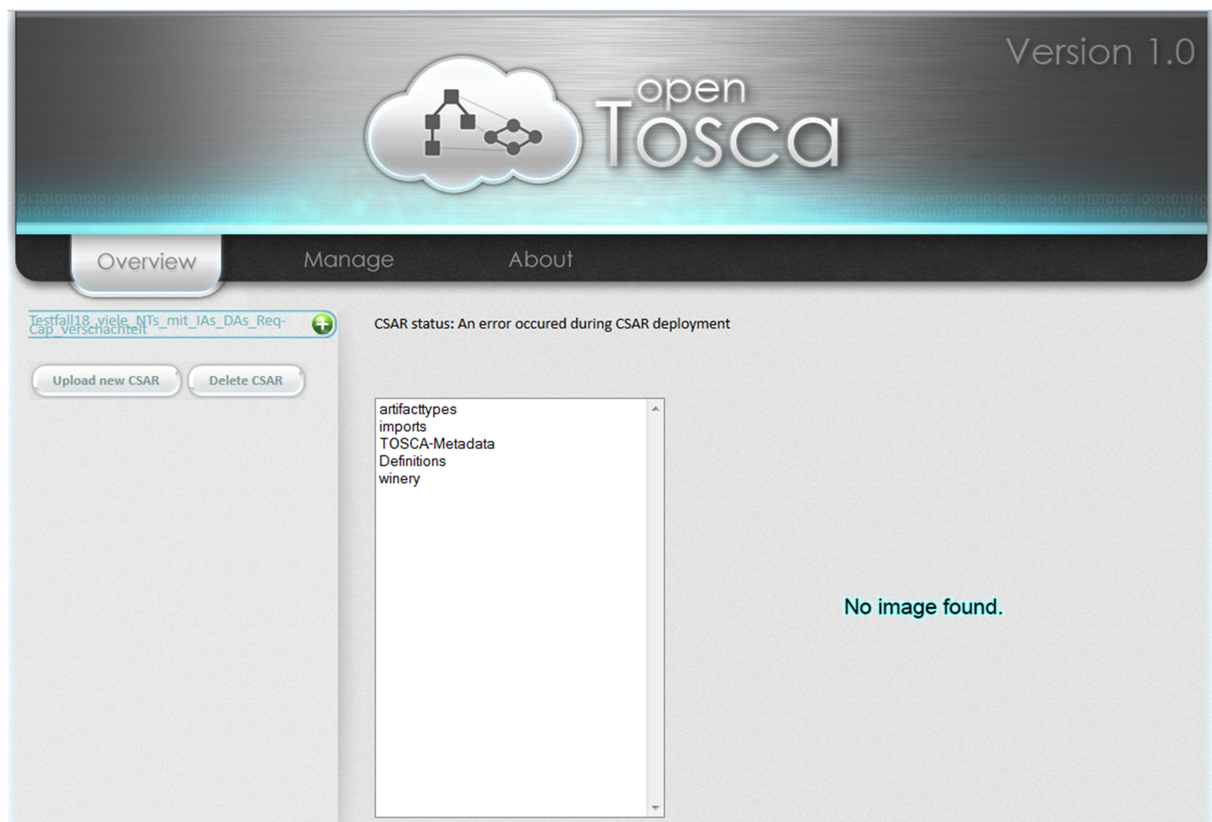

Abbildung 27: Testfall 18 erfolgreich geladen

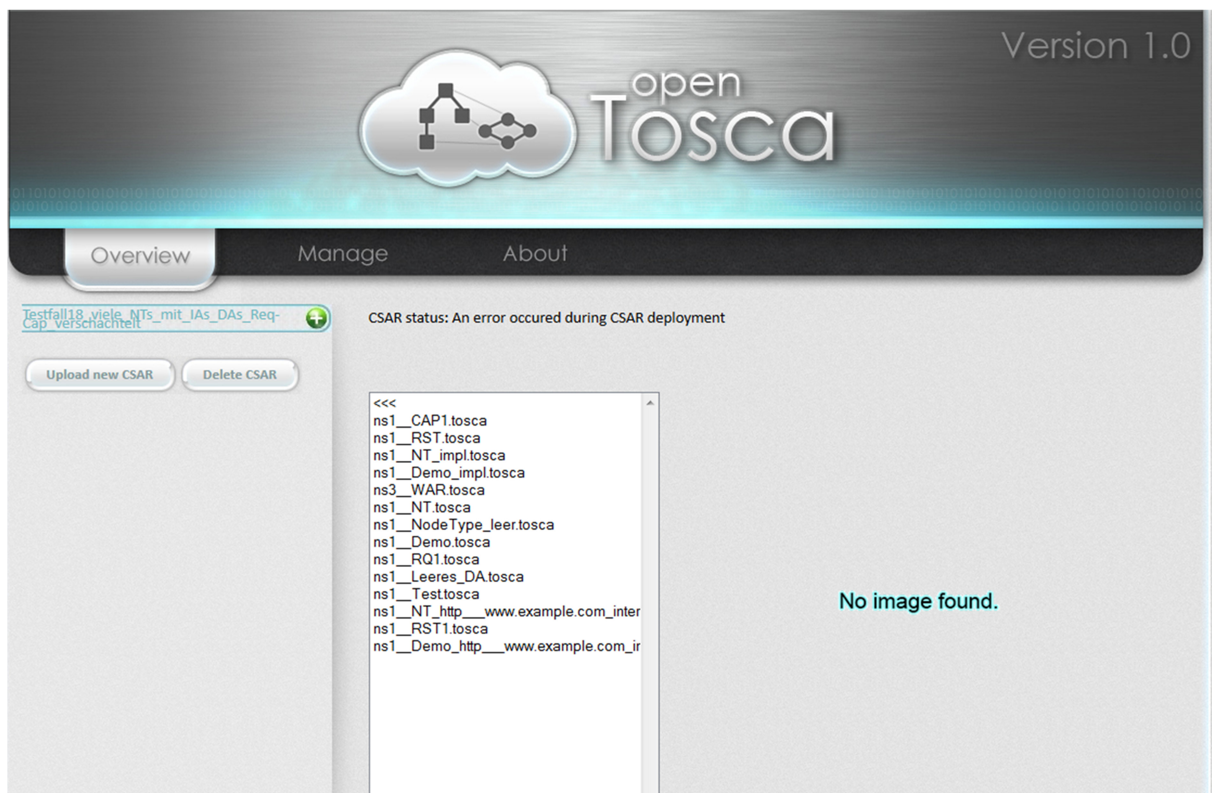

Abbildung 28: Inhalt vom geladenen Testfall 18

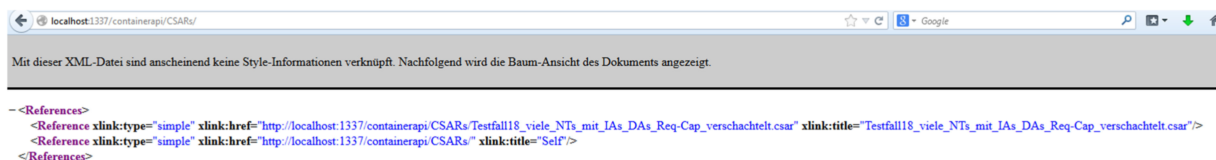

Abbildung 29: Testfall 18 im Container API der OpenTOSCA erfolgreich installiert

Bei der Ausführung der zweiten Testgruppe „Deployment-Testfälle" durch die OpenTOSCA wird eine Instanz vom Service auf der Zielumgebung instanziiert. Als Erstes werden die CSARs mithilfe der grafischen Oberfläche der OpenTOSCA (siehe Abb. 30) einzeln geladen. Wenn die CSAR erfolgreich geladen ist, startet die Komponente TOSCA Engine mit der Verarbeitung der Dokumente vom Ordner „Definitions" der CSAR. Die CSARs dieser Testgruppe haben alle einen Ordner namens „Imports", in dem sich alle Dateien, die in einem Definitions-Dokument importiert werden müssen, befinden. Anschließend werden die Dokumente der Ordner „Definitions", die bereits verarbeitet wurden, im Repository der Core-Komponente abgelegt. Im nächsten Schritt werden die Implementation Artefakte durch die IA Engine-Komponente ausgeführt. Das ist insofern wichtig, weil dadurch jetzt alle Operationen des Implementation Artefakts dem Plan zur Verfügung stehen. Hierfür werden Daten, die gebraucht werden, durch die TOSCA Engine-Komponente bezogen. Darauffolgend werden die Endpunkte des ausgeführten Artefakts in der Core-Komponente abgelegt. Zum Schluss

wird der BPEL-Plan durch die Plan Engine-Komponente ausgeführt. Die Ausführung der Pläne findet auf dem lokalen Business Prozess Server WSO2 (BPS) statt. Hierzu werden Anweisungen, die das Implementation Artefakt vom File Service aufrufen, durch den Plan realisiert. Dabei werden die jeweiligen Operationen gebunden und bekommen ihre Endpunkte, die bereits in der Core-Komponente gespeichert wurden. Mit diesem Schritt wird die Ausführung der CSARs auf der OpenTOSCA abgeschlossen und die Überprüfung kann dann begonnen werden.

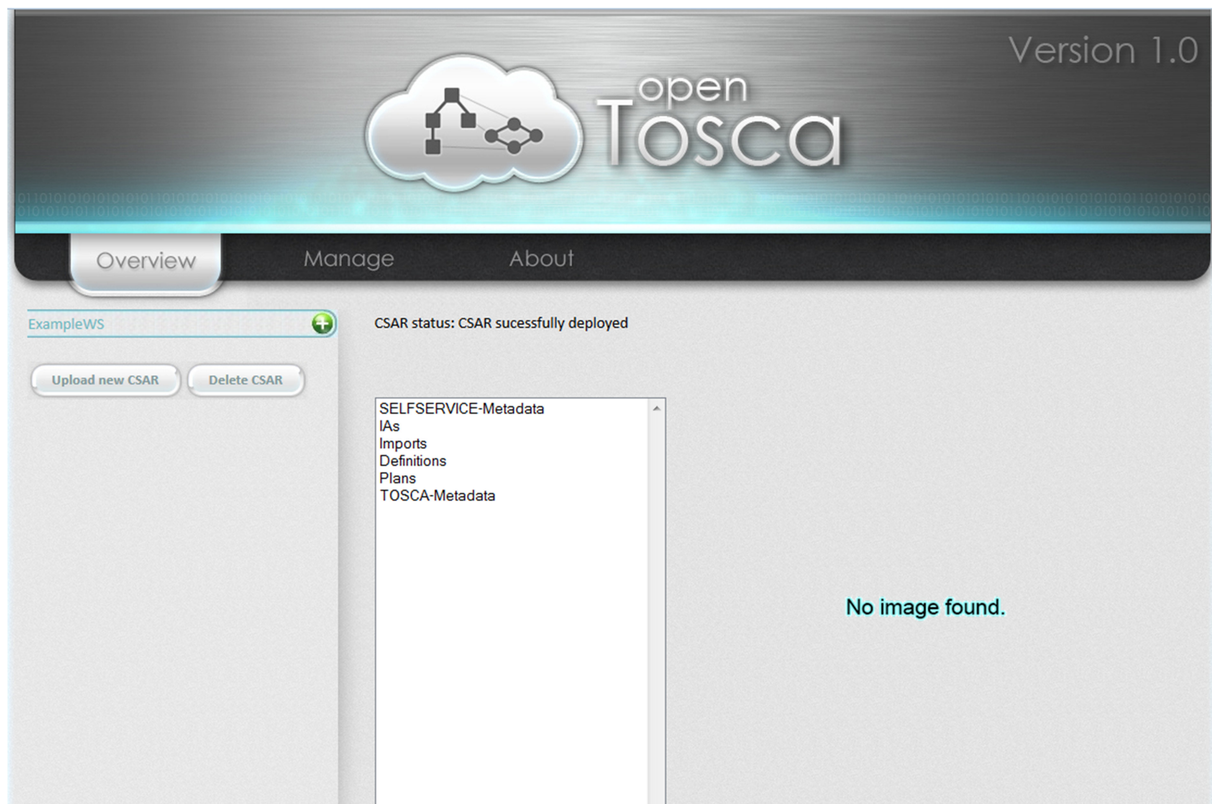

Abbildung 30: eine CSAR-Datei aus der Testfallgruppe „Deployment Testfälle" erfolgreich geladen

Im OpenTOSCA Container API kann zunächst, wie in Abbildung 31 und Abbildung 32 zu sehen ist, überprüft werden, ob die CSAR erfolgreich geladen und installiert wurde. Darüber hinaus wurde in den Logfile-Dateien der OpenTOSCA sowie auch der Tomcat nachgesehen, ob mögliche Probleme oder Fehlermeldungen vorhanden sind. Dort wurde das erfolgreiche Laden und Ausführen der CSARs bestätigt.

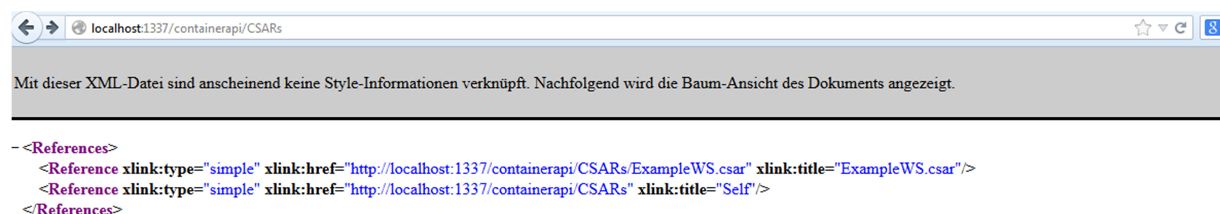

Abbildung 31: Die CSAR ExampleWS erfolgreich im Container API der OpenTOSCA installiert

Abbildung 32: Inhalt der CSAR ExampleWS im Container API der OpenTOSCA

Darauffolgend wurde im Web-Applikation-Server Tomcat7 kontrolliert, ob das Implementation Artefakt tatsächlich ausgeführt und installiert wurde. Wie man es in Abbildung 33 gut erkennen kann, wurde die WAR-Datei, die den Namen Example_WS trägt, erfolgreich in der Tomcat7 installiert.

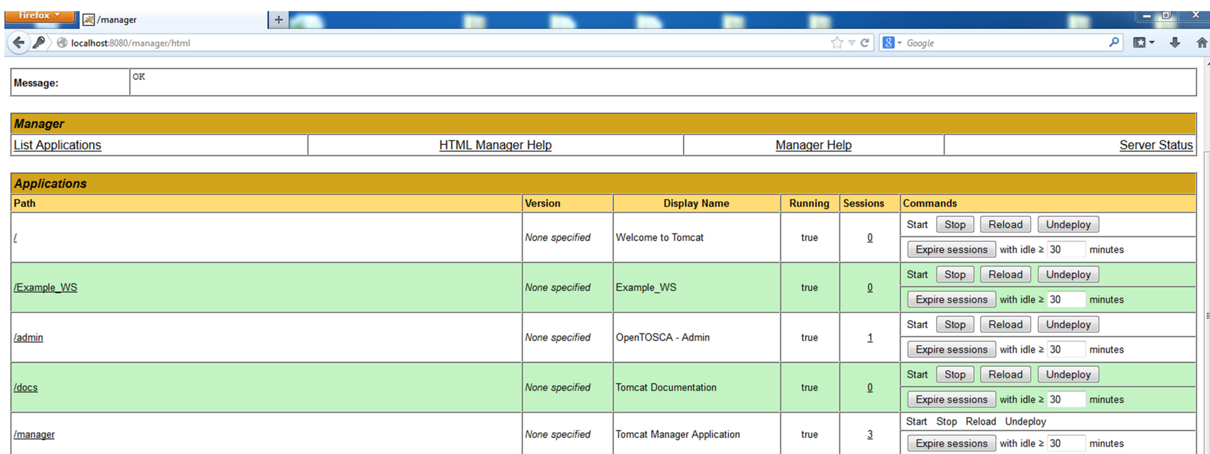

Abbildung 33: Implementation Artefakt Example_WS erfolgreich in die Tomcat installiert

Als Nächstes gilt es zu überprüfen, ob der Plan durch die OpenTOSCA Engine auf dem Business Prozess Server WSO2 BPS tatsächlich ausgeführt wurde. Dafür wird in der WSO2 BPS Benutzeroberfläche nachgesehen, ob hier ein neuer Plan zur Verfügung steht. In diesem Plan wird die Funktion Try-it aufgerufen (siehe Abb. 34), die selbst ausgeführt werden muss, damit eine neue Instanz erzeugt wird (siehe Abb. 35). Mit diesem letzten Schritt wäre die Ausführung des Plans bzw. des CSARs erfolgreich abgeschlossen.

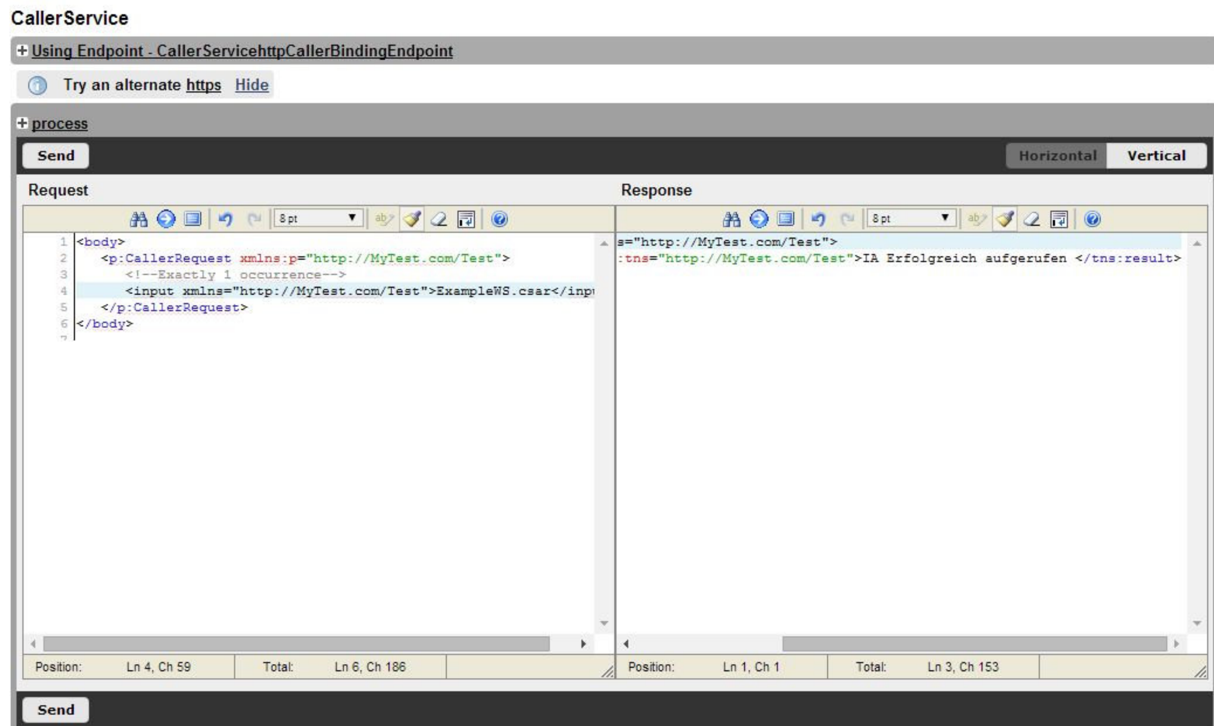

Abbildung 34: ausgeführter Plan in WSO2 BPS

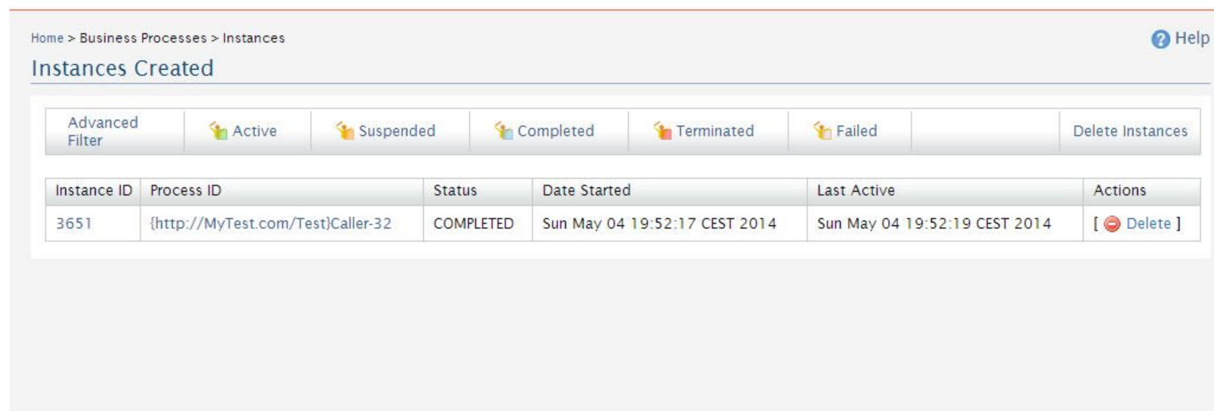

Abbildung 35: die neu erzeugte Instanz in WSO2 BPS

5.4 Analyse der Testdurchführung

Aus der in den vorhergehenden Kapiteln dargelegten Testdurchführung lassen sich folgende Erkenntnisse ableiten:
Offensichtlich kann die OpenTOSCA-Engine immer ein Implementation Artefakt durchführen, dabei erlaubt sie auch mehrere Instanzen vom selben Artefakt, die auf dem WSO2 BPS unter einem leicht veränderten Namen bzw. Nummer registriert werden, zu erzeugen. Dies kann dennoch dazu führen, dass den Überblick darüber zu behalten, welches Implementation Artefakt zu welcher Anwendung gehört, erschwert wird.

Weiter, beim Ausführen von CSARs in die OpenTOSCA-Engine müssen manche Schritte, wie das Speichern und Ausführen von BPEL-Plänen auf dem Business Prozess Server WSO2 BPS manuell gestartet werden. Es ist nicht möglich, diese Schritte automatisch durchführen zu lassen.

Hingegen erfolgte das Laden der CSAR-Dateien durch die OpenTOSCA Engine recht schnell. Die Implementation Artefakte wurden erwartungsgemäß auf dem Web-Applikation-Server Tomcat7 gespeichert. Auch die BPEL-Pläne wurden, wie zu erwarten, auf dem Business Prozess Server WSO2 BPS erfolgreich ausgeführt.

Auf Grundlage der für diese Arbeit ausgeführten Tests kann man durchaus behaupten, dass die Laufzeitumgebung OpenTOSCA eine sehr gute Leistung für nicht sehr große CSAR-Dateien gezeigt hat.

Mittlerweile wurde die Software Winery zur grafischen Modellierung von Service Templates entwickelt. Sie ermöglicht nicht nur eine grafische Erstellung von Topology Templates, die Anwendungen repräsentieren, sondern bietet ebenfalls eine automatische Generierung von CSAR-Dateien, die dann exportiert werden können. Dadurch wird der Aufwand zur Erstellung einer CSAR-Datei erheblich reduziert. Allerdings müssen die Artefakte und die Pläne nach wie vor manuell erzeugt werden und in die Winery geladen und angehängt werden. Außerdem besteht weiterhin die Möglichkeit, die Erstellung der Service Templates manuell in XML durchzuführen.
Eine Einschränkung von der OpenTOSCA ist dennoch, dass sie nur folgende drei Schritte leisten kann, nämlich eine CSAR-Datei zu laden, die Implementation Artefakte auf dem Web-Applikation-Server auszuführen und die BPEL-Pläne auf dem lokalen Business Prozess Server WSO2 BPS zu installieren.
Für das Bearbeiten der Implementation Artefakte benötigt OpenTOSCA einen lauffähigen Web-Applikation-Server Tomcat7 mit einem installierten Apachen-Axis2-Web-Applikation. Darüber hinaus müssen der Web-Service sowie die WAR-Datei innerhalb einer CSAR-Datei einmalige Namen tragen. Ansonsten läuft man Gefahr, bereits ausgeführte Dienste oder WAR-Dateien, die denselben Namen tragen, zu überschreiben.
Des Weiteren erschweren einige Einschränkungen, dass die OpenTOSCA unterschiedliche Service Templates frei kombinieren kann. So ist es in einigen Fällen nicht klar, welche Namespaces oder welche Parameter genutzt werden können. Zudem kann es durchaus vorkommen, dass die gleichen Komponenten, die in den jeweiligen Service Templates einmalig sind, fälschlicherweise mehrfach ausgeführt werden. Dies ist darauf zurückzuführen, dass die OpenTOSCA nicht überprüfen kann, ob eine Komponente bereits verfügbar ist.
Abschließend lässt sich feststellen, dass die OpenTOSCA wenig benutzerfreundlich ist, da einiges manuell gemacht werden muss. Verbesserungen könnten sowohl in Bezug auf die Benutzeroberfläche als auch bei der Automatisierung durchgeführt werden.

6 Zusammenfassung und Ausblick

Bisher konnten TOSCA-Laufzeitumgebungen nur durch vereinzelte CSAR-Dateien getestet werden. Im Rahmen dieser Arbeit wurde eine Testumgebung entwickelt, die eine große Menge an CSAR-Dateien, welche ihrerseits gemäß dem TOSCA-Standard erzeugt wurden, beinhaltet. Diese CSARs haben einen hohen Abdeckungsgrad in Bezug auf die TOSCA-Spezifikation. Damit ist es jetzt möglich jede TOSCA-Laufzeitumgebung auf ihre Kompatibilität und Compliance zu überprüfen. Darüber hinaus kann diesbezüglich ein Vergleich zwischen zwei TOSCA-Laufzeitumgebungen vollzogen werden.
Zusammenfassend kann man sagen, dass beide Gruppen der Testfälle erfolgreich entwickelt, implementiert und ausgeführt wurden. Die CSAR-Dateien wurden in die OpenTOSCA mit Erfolg geladen und installiert. Die Ausführung der Gruppe „Deployment-Testfälle“ erfolgte logischerweise nicht ganz automatisch, da die Ausführung aller BPEL-Pläne manuell gestartet wurde.
Insgesamt kann über die OpenTOSCA-Engine gesagt werden, dass sie in der Lage ist, die Erwartungen in Bezug auf die Kompatibilität und Compliance gemäß dem TOSCA-Standard zu erfüllen. Es kann davon ausgegangen werden, dass sie jede Anwendung in Form einer CSAR-Datei, die die TOSCA-Spezifikation gerecht ist, erfolgreich laden und ausführen kann. Dabei werden folgende Schritte bei der Ausführung der CSARs durch die OpenTOSCA durchgeführt:

- Das Entpacken der CSAR-Datei, sowie die Ermittlung der entpackten Ordner.
- Die Validierung des Inhalts der CSAR und der TOSCA-Metadatei.
- Das Speichern der Metadaten der CSAR auf der lokalen Datenbank.

Ein möglicher Vorschlag für künftige Arbeiten wäre, die manuelle Ausführung der Testfälle der in dieser Arbeit entwickelten Testumgebung zu automatisieren. Besonders bei häufiger Wiederholung der Testdurchführung (Regressionstest) stellt sich bei der Testautomatisierung ein deutlicher Nutzen-Faktor dar. Vor allem könnten die Automatisierung der Testdurchführung der TOSCA-Laufzeitumgebung und die anschließende Auswertung lohnenswert sein.
Ähnlich wie beim manuellen Test stehen beim automatisierten Test im Allgemeinen mehrere Schnittstellen zur Verfügung, über die ein Testobjekt getestet werden kann. Denn die Testautomatisierung muss bei der Durchführung in der einen oder anderen Form mit dem Testobjekt interagieren. Dies kann über unterschiedliche Schnittstellen der TOSCA-Laufzeitumgebung wie die GUI oder über Webservice-Schnittstellen realisiert werden. Es ist von besonderer Wichtigkeit, dass diese Schnittstellen der Automatisierung zur Verfügung stehen, um Testabläufe durchzuführen und die Reaktion des Testobjekts auszuwerten. Jedoch ist die GUI-Schnittstelle für einen Benutzer und nicht für Skripte oder Programme vorgesehen. Damit sind in der Regel auch keine Mechanismen vorhanden, um diese Schnittstellen automatisiert anzusteuern. Hierfür könnten Automatisierungswerkzeuge, die die Rolle einer Schnittstelle zum Testobjekt übernehmen, eingesetzt. Außerdem stehen der Automatisierung selbst mehrere Werkzeuge und Hilfsmittel zur Verfügung, deren Verwendung ein Automatisierungssystem effektiver machen kann [SBB11].
In Abbildung 36 ist ein Graph zu sehen, der einen allgemeinen Regressionstest illustriert. Speziell für die entwickelte Testumgebung dieser Arbeit, wurde ein Automationskonzept entwickelt, welches in Abbildung 37 anschaulich dargestellt wird.

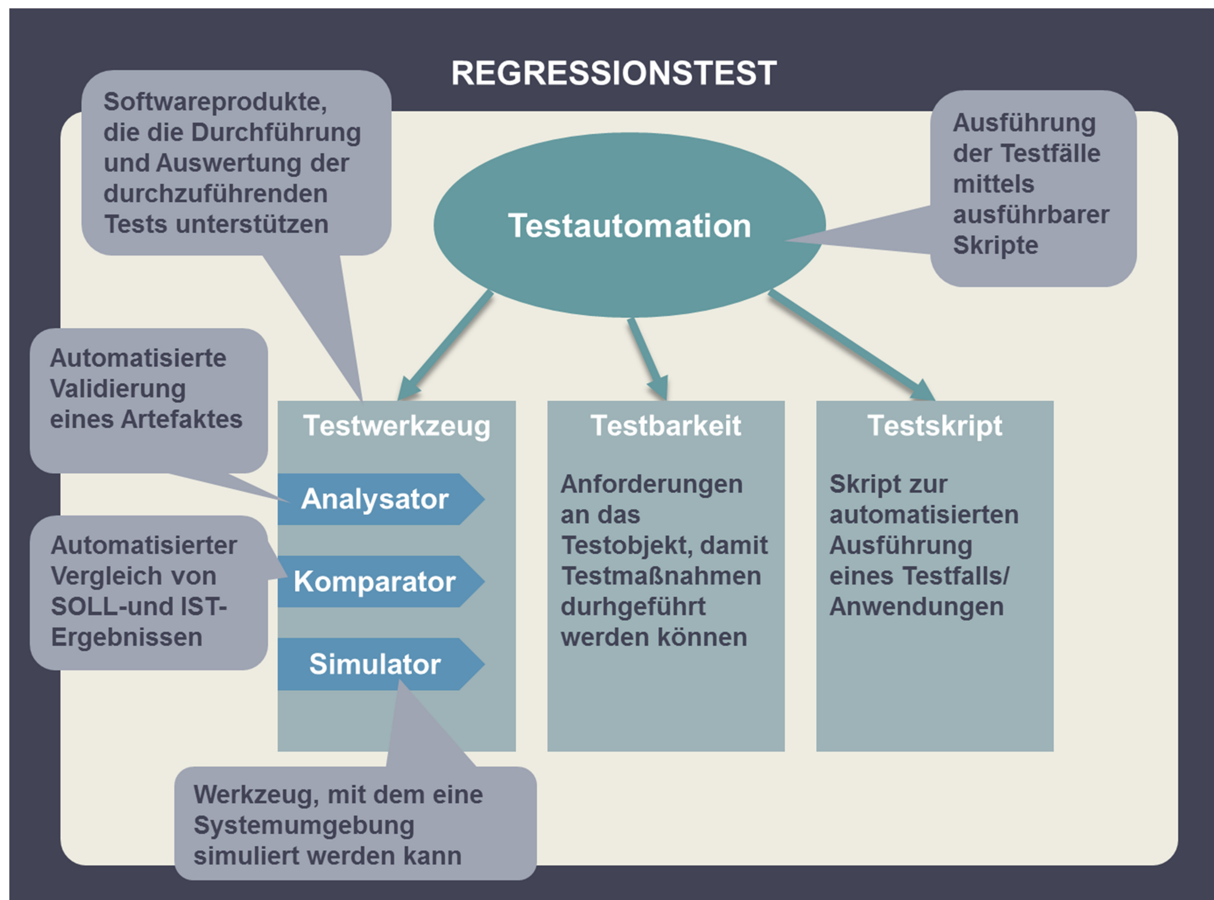

Abbildung 36: Regressionstest (Automation Allgemein)

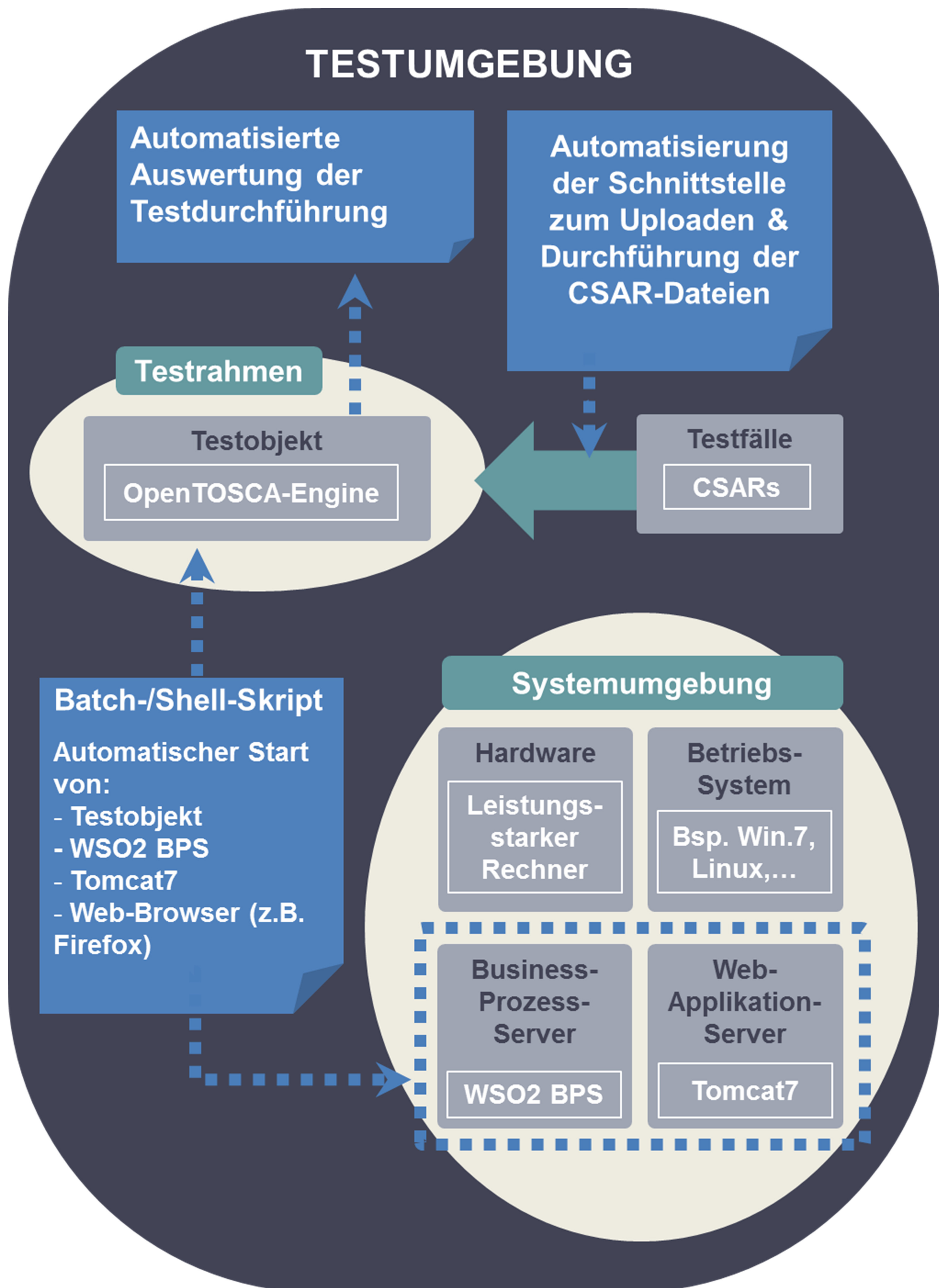

Abbildung 37: Automation der Testumgebung

Beschreibung:

- Als Erstes soll mithilfe von Batch- bzw. Shell-Skripten (je nach Betriebssystem) ein automatisiertes Starten des Web-Applikation-Servers Tomcat7, des Business Prozess Servers WSO2 BPS, des Testobjekts (hier OpenTOSCA) und eines Web-Browsers (Firefox) programmiert werden.
- Das Laden und Ausführen der CSAR-Dateien muss automatisiert werden. Dies erfordert die Entwicklung einer Lösung, die diese Schnittstelle implementiert.
- Zur Automatisierung der Testauswertung muss das erhaltene Testergebnis mit dem Erwartungswert verglichen werden. Darüber hinaus sollte aus den erhaltenen Testergebnissen und mithilfe von z. B. Dokumentengeneratoren eine verständliche Metrik, die die Anzahl erfolgreicher Testfälle zeigt, automatisch erzeugt werden.

Literaturverzeichnis

[SL11] Spillner, Andreas; Linz, Tilo: Basiswissen Softwaretest, dpunkt, 2011

[SLRW07] Spillner, Andreas; Linz, Tilo; Roßner, Thomas; Winter, Mario: Basiswissen Softwaretest: Testmanagement, dpunkt, 2007

[TOSCA13] Topology and Orchestration for Cloud Applications (TOSCA), Version 1.0, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf

[TOS13] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0, 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.pdf

[CC12] CloudCycle Plattform, 2012. URL: http://www.cloudcycle.org/

[XML01a] XML Schema deutsche Übersetzung Teil 0: Einführung, 2001.
URL: http://www.edition-w3.de/TR/2001/REC-xmlschema-0-20010502/

[XML01b] XML Schema deutsche Übersetzung Teil 1: Struktur, 2001.
URL: http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XML01c] XML Schema deutsche Übersetzung Teil 2: Datentypen, 2001.
URL: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[SD08] Schumm, David; Graphische Modellierung von BPEL Prozessen unter Verwendung von BPMN Notation, 2008. URL: http://www.iaas.uni-stuttgart.de/institut/ehemalige/schumm/DIP-2720.pdf

[BPE07] Web Service Business Process Execution Language, 2007.
URL: http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf

[Ope12] Openstack Foundation: Open Source Platform for building private and public Cloud. URL: http://docs.openstack.org/index.html

[OAS12] OASIS TOSCA TC. Topology and Orchestration Specification for Cloud Applications, working draft 05, 2012. URL: https://www.oasis-open.org/committees/document.php?document_id=45638&wg_abbrev=tosca

[Wie13] Wieland, Matthias; Ausschreibung der Diplomarbeit, 2013.

[Ope13] OpenTOSCA: Institut für Architektur von Anwendungssystemen (IAAS), 2013. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/

[KBBL13] Kopp, Oliver; Binz, Tobias; Breitenbücher; Leymann, Frank; Winery- A Modelling Tool for TOSCA-Based Cloud Applications, 2013
URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2013-46%20-%20Winery_A_Modeling_Tool_for_TOSCA-based_Cloud_Applications.pdf

[NAT11] National Institute of Standards and Technology. The NIST Definition of Cloud Computing, 2011.
URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[COM11] Computerbild. CeBIT-Lexikon: Fachbegriffe und Abkürkzungen leicht verständlich erklärt, 2011. URL: http://www.computerbild.de/artikel/cb-Messe-CeBIT-PC-Fachbegriffe-CeBIT-Lexikon-2308635.html

[OAS07] OASIS WSBPEL TC: Web Services Business Process Execution Language version 2 Specification, 2007

[WSO13] WSO2 Business Process Server Documentation Plattform, 2013.
URL:https://docs.wso2.org/display/BPS300/WSO2+Business+Process+Server+Documentation

[TU10] TU-Dresden; Java-basierten Web Service erstellen, Tutorial, 2010.
URL: http://www.rn.inf.tu-dresden.de/lectures/EvSaBSA/Tutorial%201%20-%20Java-basierten%20Web%20Service%20erstellen.pdf

[SBB11] Seidl, Richard; Baumgartner, Manfred; Bucsics, Thomas: Basiswissen Testautomatiserung, dpunkt, 2011

Anhang

In der untensteheden Abbildung 38 wird eine allgemeine Übersicht über die Struktur der mitgelieferten CD dargestellt. Der Inhalt umfasst alle wesentlichen Dokumente, Testfälle und Softwares, die im Rahmen dieser Diplomarbeit entstanden und verwendet wurden.
An dieser Stelle sei darauf hingewiesen, dass in den jeweiligen Ordnern eine Textdatei „Information“ zu finden ist. Diese beinhalten eine detaillierte Beschreibung vom Inhalt der jeweiligen Ordner.

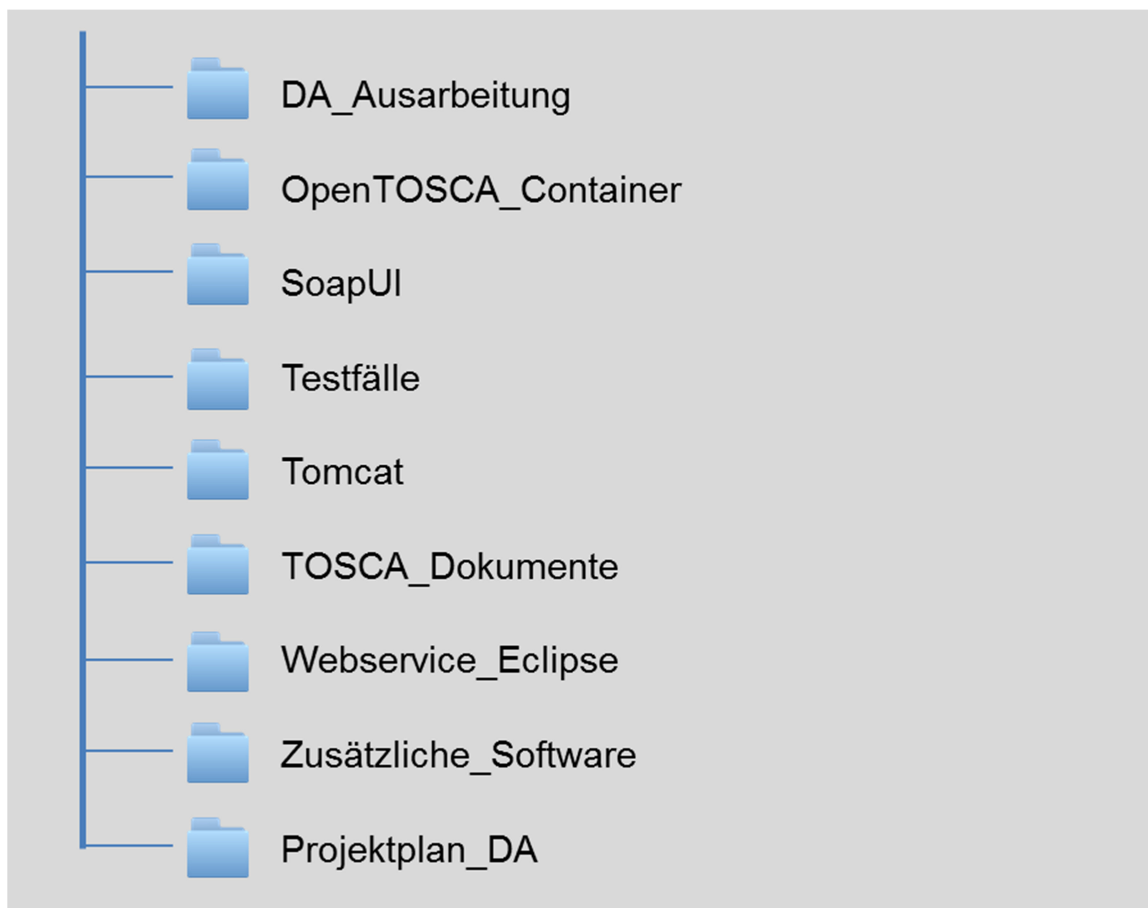

Abbildung 38: Struktur der mitgelieferten CD

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 22. Mai 2014 ______________________