Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3510

Automatische Vervollständigung von Topologien für TOSCA-basierte Cloud-Anwendungen

Pascal Hirmer

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr. Frank Leymann

Betreuer/in: Dipl.-Inf. Uwe Breitenbücher

Beginn am: 17. Juni 2013

Beendet am: 17. Dezember 2013

CR-Nummer: K.1, K.6.4, D.2.12

Kurzfassung

Seit einigen Jahren sind Cloud Computing-Technologien weit verbreitet. Statt Anwendungen, Rechenprozesse, Datenspeicherung oder ähnliche IT-Dienste auf lokalen Rechnern oder Servern auszuführen, werden diese sowohl von Firmen als auch von Privatpersonen vermehrt in Clouds ausgelagert. Dies bietet viele Vorteile wie Skalierbarkeit, Kostenreduzierung und eine hohe Verfügbarkeit von Anwendungen. Um eine Anwendung automatisiert in die Cloud auszulagern, werden häufig Topologien erstellt, die alle Komponenten und Relationen beschreiben, die für eine Provisionierung notwendig sind. Diese können mit Hilfe des OASIS-Standards TOSCA (Topology and Orchestration Specification for Cloud Applications) [1] modelliert werden. Um vollständige TOSCA-Topologien für eine Provisionierung zu erstellen, müssen jedoch sehr viele Details (Anbieter, Web Server-Versionen etc.) beschrieben werden. Haben Entwickler von Cloud-Anwendungen keine konkreten Anforderungen an Providerwahl, Infrastruktur oder Plattformen, sollten ihnen derartige Detail-Entscheidungen abgenommen werden. Im Rahmen dieser Diplomarbeit wird ein Konzept entwickelt und implementiert, welches es dem Anwendungsentwickler ermöglicht, unvollständige Topologien zu modellieren, die lediglich anwendungsspezifische Komponenten und Relationen enthalten. Diese werden anschließend automatisch durch das Hinzufügen von Infrastruktur- und Plattform-Komponenten, mit der im Rahmen dieser Arbeit entwickelten Lösung, zu einer provisionierbaren Topologie vervollständigt. Zusätzlich wird die Möglichkeit geschaffen eine assistierte Modellierung durchzuführen. Bei dieser werden dem Modellierer in jedem Schritt der Modellierung - zur Topologie passende - Komponenten und Relationen vorgeschlagen, aus denen er auswählen kann. Die ausgewählten Komponenten und Relationen werden anschließend der Topologie hinzugefügt. Dies hilft insbesondere ungeübten Modellierern eine TOSCA-Topologie korrekt zu vervollständigen. Um die entworfenen Konzepte in der Praxis anwendbar zu machen, werden diese in das grafische TOSCA-Modellierungstool Winery [2] integriert.

Inhaltsverzeichnis

1. Einleitung **11**
- 1.1. Motivation und Zielsetzung . . . 11
- 1.2. Aufbau und Kapitelübersicht . . . 12
- 1.3. Begriffserläuterungen . . . 12

2. Grundlagen **13**
- 2.1. Cloud Computing . . . 13
- 2.2. Topologie-Graphen . . . 15
- 2.3. TOSCA - Topology and Orchestration Specification for Cloud Applications . . . 16
  - 2.3.1. Cloud Service Archive (CSAR) . . . 22
  - 2.3.2. Requirements und Capabilities . . . 23
    - 2.3.2.1 Anwendungsbeispiel: Verwendung von Requirements . . . 24
    - 2.3.2.2 Matching von Requirement und Capability . . . 24
  - 2.3.3. Node Template-Ersetzung . . . 25
  - 2.3.4. Abstrakte Typen . . . 26
  - 2.3.5. TOSCA und Policys . . . 26
  - 2.3.6. Winery - Ein grafisches TOSCA-Modellierungstool . . . 27

3. Problembeschreibung - Vervollständigung von TOSCA-Topologien **29**
- 3.1. Aufgabenstellung . . . 29
- 3.2. Herausforderungen . . . 30

4. Konzeptionelle Lösung **33**
- 4.1. Vervollständigung einer Topologie in einem Schritt . . . 33
- 4.2. Assistierte Modellierung vollständiger Topologien . . . 33
- 4.3. Anwendungsfallanalyse: Modellierung unvollständiger Topologien . . . 34
  - 4.3.1. Anwendungsfall 1: Unvollständige TOSCA-Topologien mit Requirements 35
  - 4.3.2. Anwendungsfall 2: Unvollständige TOSCA-Topologien ohne Einschränkungen . . . 37
  - 4.3.3. Anwendungsfall 3: Teilmodellierung der Infrastruktur . . . 39
  - 4.3.4. Anwendungsfall 4: Vereinfachte Teilmodellierung der Infrastruktur . . . 43
    - 4.3.4.1 Abgrenzung zu Anwendungsfall 3 - Platzhalter-Modellierung . . . 45
  - 4.3.5. Kombination der Anwendungsfälle . . . 46
  - 4.3.6. Zusammenfassung . . . 47
- 4.4. Vervollständigung von TOSCA-Topologien . . . 48
  - 4.4.1. Anforderungen und Voraussetzungen . . . 48
  - 4.4.2. Type-Repository . . . 48

4.4.3. Algorithmen der Topologie-Vervollständigung . . . 50
4.4.3.1 Hilfsalgorithmen . . . 50
4.4.3.2 Vervollständigungs-Modi . . . 52
4.4.3.3 Grundkonzept: Untersuchung auf Provisionierbarkeit . . . 53
4.4.3.4 „TopologyCompletion“ - Hauptalgorithmus der Vervollständigung 54
4.4.3.5 Algorithmus „FulfillRequirements“ . . . 57
4.4.3.6 Algorithmus „ReplacePlaceholders“ . . . 61
4.4.3.7 Algorithmus „ProcessDeferredRelationshipTemplates“ . . . 64
4.4.4. Auswahl der Templates . . . 68
4.4.4.1 Auswahl der Node Templates . . . 68
4.4.4.2 Auswahl von Relationship Templates . . . 69
4.4.5. Schrittweise Vervollständigung . . . 69
4.5. Integration der Konzepte in Winery . . . 70

5. Umsetzung **71**
5.1. Entwurf und Architektur-Design . . . 71
5.1.1. Anwendungsfälle . . . 72
5.1.2. Komponenten der Topologie-Vervollständigung . . . 76
5.1.2.1 TopologyCompleter . . . 78
5.1.2.2 TopologyAnalyzer-Komponente . . . 78
5.1.2.3 Topology Completion-Komponente . . . 79
5.1.2.4 RepositoryConnector-Komponente . . . 80
5.1.2.5 Utils-Komponente . . . 81
5.1.3. Sequenz Diagramme zur Topologie-Vervollständigung . . . 82
5.1.4. Winery Topology Modeler-Erweiterung . . . 84
5.1.4.1 Erweiterung bestehender Winery-Komponenten . . . 84
5.1.4.2 Neu geschaffene Winery-Komponenten (JSPs) . . . 85
5.2. Implementierung und Ergebnis . . . 87
5.2.1. Vervollständigung einer Topologie in einem Schritt . . . 87
5.2.2. Schrittweise Vervollständigung einer Topologie / Assistierte Modellierung . . . 93
5.2.3. Überprüfen einer Topologie auf Provisionierbarkeit . . . 95
5.2.4. Suchen eines Container-Node Templates . . . 95
5.2.5. Sonderfälle . . . 97

6. Verwandte Arbeiten **99**
6.1. Provisionierung von TOSCA-Topologien . . . 101

7. Fazit und Ausblick **103**
7.1. Fazit . . . 103
7.2. Ausblick . . . 104
7.2.1. TOSCA-Vervollständigung mit Policys . . . 104
7.2.2. Eigenschaften von TOSCA-Templates . . . 104

A. Anhang 105
A.1. XML-Code der Beispiel TOSCA-Topologie aus Abbildung 2.5 105
A.2. Typ-Umfang der Testfälle 106
A.3. TOSCA-Policy Format 109
A.3.1. Policys im Vervollständigungs-Algorithmus 111

Literaturverzeichnis 113

Abbildungsverzeichnis

2.1. Übersicht Cloud-Service-Modelle (angelehnt an [3]) 14
2.2. Grafische TOSCA-Notation (aus [4] bzw. [5]) 17
2.3. TOSCA Überblick (angelehnt an [4]) 18
2.4. Vererbung von TOSCA-Typen 18
2.5. Grafische TOSCA Beispiel-Topologie 21
2.6. Ordnerstruktur CSAR [4] 22
2.7. TOSCA Requirements und Capabilitys (angelehnt an [4]) 23
2.8. Beispiel für die Verwendung von Requirements 24
2.9. Ersetzung von Node Templates (Node Template Substitution) [4] 25
2.10. Topologie-Modellierung in Winery 27
2.11. Oberfläche des Winery Repositorys 28

4.1. Beispiel-Topologie nach Anwendungsfall 1 36
4.2. Beispiel-Topologie nach Anwendungsfall 2 38
4.3. Beispiel: Vererbung von Platzhaltern 40
4.4. Beispiel-Topologie nach Anwendungsfall 3 42
4.5. Verwendung von Relationship Templates des Typs „Deferred“ 44
4.6. Mögliche Kombination der Anwendungsfälle 46
4.7. Winery Repository 49
4.8. Winery Integration 70

5.1. Use Cases: Topologie-Vervollständigung 72
5.2. Komponentenmodell der Topologie-Vervollständigung 76
5.3. Klassen der Topologie-Vervollständigung 77
5.4. Ablauf der Topologie-Vervollständigung 82
5.5. Ablauf der Topologie-Vervollständigung mit Nutzerinteraktion 83
5.6. Winery Komponenten-Übersicht (auf Basis von [2]) 84
5.7. Abhängigkeiten des Winery Topology Modelers 86
5.8. Modellierung einer unvollständigen TOSCA-Topologie 88
5.9. Topologie-Vervollständigungs-Dialog 88
5.10. Dialog zur Auswahl einer Topologie-Lösung 90
5.11. End-Resultat der Topologie-Vervollständigung 91
5.12. Dialog zur Auswahl von Relationship Templates 92
5.13. Auswahl von Node und Relationship Templates beim Step-by-Step-Ansatz . . 94
5.14. Provisionierbarkeits-Überprüfung 95
5.15. Auswahl eines Containers 96

A.1. Typ-Umfang Node und Relationship Types 107
A.2. Typ-Umfang Requirement und Capability Types 108

Tabellenverzeichnis

2.1. Aufbau eines TOSCA-Definitions-Dokuments 19

Verzeichnis der Listings

2.1. TOSCA XML Aufbau 20

A.2. Policy Type für die Vervollständigung 110
A.3. Policy Template für die Vervollständigung 111

Verzeichnis der Algorithmen

4.1. Pseudo-Code: Suche passende Verbindungen zweier Node Templates 51
4.2. Pseudo-Code: Instanziierung von Templates 52
4.3. Pseudo-Code: Algorithmus „TopologyCompletion“ 56
4.4. Pseudo-Code: Analyse von Requirements 57
4.5. Pseudo-Code: Matching von Requirements und Capabilities 58
4.6. Pseudo-Code: Algorithmus „FulfillRequirements“ 60
4.7. Pseudo-Code: Analyse der Platzhalter 61
4.8. Pseudo-Code: Algorithmus SearchPlaceholderReplacement 62
4.9. Pseudo-Code: Algorithmus der Vervollständigung von Platzhaltern 63
4.10. Pseudo-Code: Analyse von Relationship Templates des Typs „Deferred“ 64
4.11. Pseudo-Code: Algorithmus der Vervollständigung von Relationship Templates des Typs „Deferred“ 67

1. Einleitung

1.1. Motivation und Zielsetzung

Cloud Computing bietet sowohl Anbietern als auch Nutzern von Cloud-Anwendungen viele Vorteile. Darunter die Reduzierung von Wartungs- und Nutzungskosten sowie eine hohe Verfügbarkeit und Skalierbarkeit. Um Anwendungen automatisiert in der Cloud provisionieren zu können, werden häufig Topologien entworfen, die alle Komponenten und Relationen beschreiben, die bei der Provisionierung einer Anwendung involviert oder erzeugt werden. Die standardisierte Modellierung solcher Topologien wird durch den OASIS[1]-Standard TOSCA[2] [1] ermöglicht. Dieser bietet den Vorteil der Wiederverwendbarkeit von Topologien sowie eine erhöhte Portierbarkeit von Cloud-Services zu unterschiedlichen Cloud-Providern [6]. Eine automatisierte Provisionierung von Anwendungen ist in einigen TOSCA-Laufzeitumgebungen (zum Beispiel OpenTOSCA[7]) jedoch nur dann möglich, wenn die Topologie vollständig modelliert wurde. Die Modellierung einer vollständigen Cloud-Topologie erfordert dabei viel Aufwand und Detail-Wissen des Modellierers. Vor allem Entscheidungen bzgl. Providerwahl, Plattformen und Infrastruktur (Cloud-Anbieter, Datenbanken, Servereinstellungen etc.) möchten Anwendungsentwickler nicht selbst treffen, wenn sie keine konkreten Anforderungen an diese haben.

Diese Diplomarbeit beschäftigt sich damit den Aufwand der Topologie-Modellierung zu reduzieren, indem dem Modellierer derartige Entscheidungen abgenommen werden. Um dies zu erreichen wird eine Möglichkeit geschaffen, lediglich eine unvollständige Topologie anfertigen zu müssen, die ausschließlich anwendungsspezifische Details enthält und die Modellierung der Infrastruktur offen lässt. Dabei kann der Modellierer funktionale Anforderungen an die Infrastruktur festlegen. Die vom Anwendungsentwickler modellierte unvollständige Topologie wird anschließend durch die, im Rahmen dieser Diplomarbeit, entwickelte Lösung zu einer provisionierbaren Topologie vervollständigt. Dabei werden dem Nutzer jeweils Lösungsvorschläge vollständiger Topologien zur Auswahl angeboten. Die auf diese Weise vervollständigte Topologie kann anschließend für eine automatisierte Provisionierung verwendet werden. Zusätzlich wird in dieser Diplomarbeit ein Konzept für eine assistierte Modellierung entwickelt, die den Modellierer schrittweise durch die Vervollständigung einer Topologie führt. Um die Vervollständigung von Topologien in der Praxis anwendbar zu machen, werden die entwickelten Konzepte in das grafische TOSCA-Modellierungstool Winery [2] integriert.

[1] Organization for the Advancement of Structured Information Standards
[2] Topology and Orchestration Specification for Cloud Applications

1.2. Aufbau und Kapitelübersicht

Dieses Dokument gliedert sich in drei Abschnitte, welche die Meilensteine Einarbeitung, Konzeptentwurf und Konzept-Umsetzung widerspiegeln. Nach den Einführungs- und Grundlagen-Kapiteln, in denen ein Überblick über Cloud-Computing, Topologien und TOSCA gegeben wird, folgt - anschließend an die Problembeschreibung - die konzeptionelle Lösung in Kapitel 4. Diese gliedert sich wiederum in zwei Teil-Abschnitte. Im ersten Abschnitt wird ein Konzept für die generische Modellierung unvollständiger Cloud-Topologien auf Basis von TOSCA beschrieben. Der zweite Abschnitt schildert ein Konzept zur Vervollständigung dieser zu provisionierbaren Topologien. Kapitel 5 beschäftigt sich anschließend mit der Umsetzung bzw. Implementierung der Konzepte und der Integration in das TOSCA-Modellierungstool Winery. Im darauffolgenden sechsten Kapitel wird auf verwandte Arbeiten hingewiesen. Abschließend wird in Kapitel 7 ein Fazit über die entstandene Arbeit sowie ein möglicher Ausblick über die zukünftige Forschung in diesem Themenbereich gegeben.
Die in dieser Diplomarbeit erstellten Grafiken wurden mit Microsoft Powerpoint 2007, dem Open Source-Zeichentool DIA[3], Microsoft Visual Studio 2012 und dem TOSCA Topology Modeler von Winery erstellt.

1.3. Begriffserläuterungen

In diesem Abschnitt werden die Begriffe Deployment und Provisionierung voneinander abgegrenzt.

- **Abgrenzung Deployment - Provisionierung**
  Die Begriffe Deployment und Provisionierung werden im Bereich des Cloud-Computings oftmals als Synonyme verwendet. In vorliegender Arbeit werden die beiden Begriffe wie folgt abgegrenzt: Unter Deployment versteht man das Hosten einer Anwendung auf einer vorhandenen Infrastruktur, unter Provisionierung wird hingegen das Hosten der gesamten Infrastruktur mitsamt der Anwendung bezeichnet. Im Rahmen dieser Arbeit steht vor allem die Provisionierung von Anwendungen im Vordergrund.

[3] https://projects.gnome.org/dia/

2. Grundlagen

In diesem Kapitel werden alle thematischen Grundlagen behandelt, die für die Nachvollziehbarkeit der Diplomarbeit notwendig sind. Weiterführende Informationen sind den jeweiligen Quellen zu entnehmen.

2.1. Cloud Computing

In diesem Abschnitt wird ein allgemeiner Überblick über das Thema Cloud Computing gegeben. Aufgrund der Vielzahl von Cloud Computing-Technologien, beschränkt sich das Einführungskapitel auf die für diese Diplomarbeit relevanten Themen.
Mit dem Begriff Cloud Computing bezeichnet man im Allgemeinen das Auslagern von IT-Diensten (Services) zu Cloud-Anbietern (Providern), die diese verteilt auf globalen Rechenzentren betreiben. Das hat für den Anwender der Services den Vorteil, dass kein Aufwand für die Wartung eigener Soft-, Middle- und Hardware besteht. Dabei werden Ort und Infrastruktur, auf denen die ausgelagerten Services betrieben werden verborgen, was zur Metapher der Cloud (dt.: Wolke) führte. Der Zugriff geschieht dabei zumeist über eine Internetverbindung. Der Cloud-Provider verpflichtet sich beim Bereitstellen von Services in sogenannten Service Level Agreements (SLA) Verfügbarkeit, Kosten sowie festgelegte Sicherheitsaspekte einzuhalten [8].

„On-Demand" ist ein Schlüsselbegriff im Cloud Computing. Darunter versteht man, dass durch die Nutzung eines Service nur dann Kosten entstehen, wenn dieser in Anspruch genommen wird. Vergleichbar ist dieses Prinzip mit Wasser- oder Stromanbietern, deren Leistungen - von der Grundgebühr abgesehen - ebenfalls nur bei Bedarf Kosten verursachen. Neben den Kostenvorteilen bieten Clouds im Gegensatz zum lokalen Betrieb von Soft-, Middle- und Hardware den Vorteil der Skalierbarkeit. Die große Anzahl global verfügbarer Rechner der Cloud-Provider macht es möglich, die Leistung eines Services nach Bedarf anzupassen. Steigt beispielsweise die Anzahl der Nutzer, können die Ressourcen spontan erhöht werden. All diese Vorteile werden insbesondere durch Virtualisierungstechnologien ermöglicht.

Im Cloud Computing existieren drei grundsätzliche Deployment-Modelle: Public-, Private- und Hybrid-Clouds. Public-Clouds sind Clouds öffentlicher Cloud-Anbieter, die prinzipiell von jeder Person, von jedem Standort aus verwendet werden können. Private-Clouds hingegen sind im „Privat-Besitz" des jeweiligen Anwenders und können nur von diesem genutzt

werden. Die Nutzung von Private-Clouds ist besonders bei der Verarbeitung sicherheitskritischer Daten sinnvoll. Eine gemeinsame Verwendung von Public- und Private-Clouds wird als Hybrid-Cloud bezeichnet. Hierbei werden sensible Daten in einer Private-Cloud, sicherheits-unkritische Daten in einer Public-Cloud gespeichert. In dieser Arbeit werden des Weiteren die Cloud-Service-Modelle „Software as a Service“ (SaaS), „Infrastructure as a Service“ (IaaS) und „Platform as a Service“ (PaaS) behandelt. Beim „SaaS“-Modell wird Software in die Cloud ausgelagert, die von einer Vielzahl von Anwendern genutzt werden kann. Ein typisches Beispiel hierfür ist die Google-Applikation „Google Docs“, eine cloud-basierte Office-Anwendung. Das „IaaS“-Modell hingegen dient der Verlagerung von Infrastruktur-Komponenten (Hardware) in die Cloud. Hierbei wird dem Nutzer Rechenleistung und Speicherplatz vom Cloud-Anbieter zur Verfügung gestellt. Der bekannteste Anbieter von IaaS-Services ist dabei die „Amazon Elastic Compute Cloud (Amazon EC 2)“[1]. Zwischen den „SaaS“- und „IaaS“-Modellen liegt das „PaaS“-Modell. Dabei handelt es sich um das Verlagern von Plattformen in die Cloud, die als Software-Laufzeitumgebung dienen. Ein Beispiel hierfür ist Amazon Beanstalk[2], das als Plattform bzw. Laufzeitumgebung für Java Web-Anwendungen genutzt werden kann. Die folgende Grafik stellt das Zusammenspiel der Cloud-Service-Modelle dar.

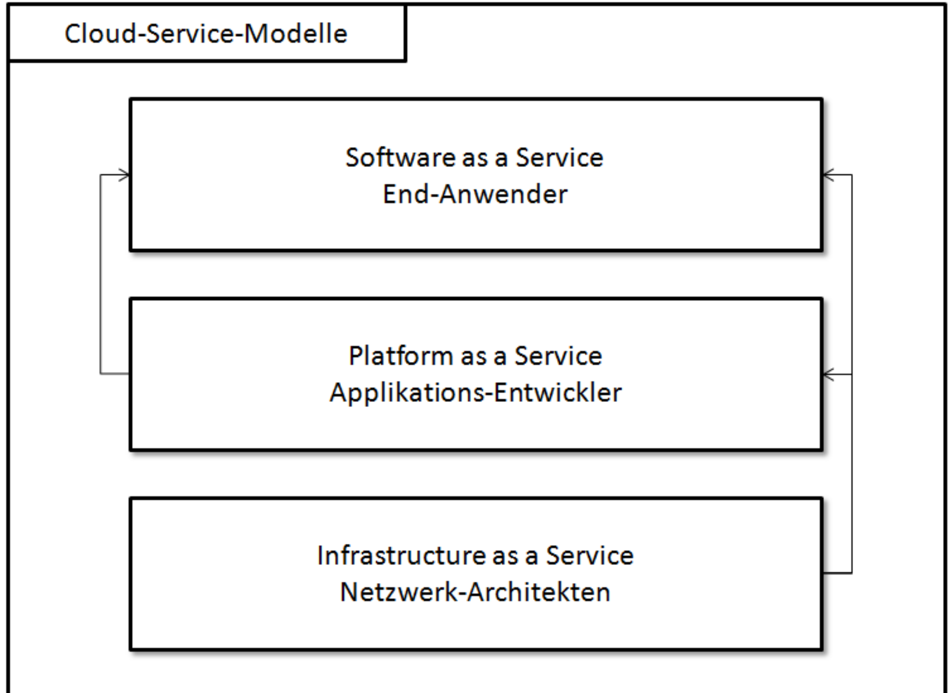

Abbildung 2.1.: Übersicht Cloud-Service-Modelle (angelehnt an [3])

[1] http://aws.amazon.com/de/ec2/
[2] http://aws.amazon.com/de/elasticbeanstalk/

2.2. Topologie-Graphen

In diesem Kapitel werden Topologien sowohl allgemein als auch bezogen auf Cloud-Computing-Systeme beschrieben. Dieser Abschnitt basiert auf [9].

Unternehmen besitzen meist umfangreiche IT-Systeme bestehend aus Infrastruktur, Software-Komponenten, Diensten, Prozessen und deren Relationen. Diese können sowohl lokal als auch in Clouds ausgelagert betrieben werden. Vor allem nach Firmen-Übernahmen, Fusionen oder Reorganisationen ist der Aufbau und das Zusammenspiel der Komponenten der Systeme oftmals nicht mehr im Ganzen ersichtlich. Dies führt zu hohem Management-Aufwand, der Gefahr von duplizierten Komponenten oder zu Systemausfällen, deren Ursache nur schwer gefunden werden kann. Änderungen an derartigen Systemen werden durch die mangelnde Übersicht fehleranfällig und kostspielig. Um diese Probleme zu verhindern, führen Binz et al. [9] sogenannte Enterprise Topologie Graphen (ETG) ein, die eine Formalisierung solcher Systeme ermöglichen und das Management dadurch vereinfachen sollen. Diese Topologie-Graphen basieren dabei auf etablierten Graphen-Theorien, wodurch bspw. die einfache Implementierung bekannter Such-Algorithmen ermöglicht wird. Durch eine einheitliche Formalisierung dieser Topologien und die dadurch entstehende Maschinen-Lesbarkeit, werden Fusionen und Reorganisationen von IT-Systemen stark vereinfacht [10].

Topologie-Graphen bestehen aus einer Knotenmenge V, die alle IT-Komponenten des Systems enthält und einer Kantenmenge E, die die Relationen (bspw. Datenverbindungen) zwischen diesen beschreibt. Bei einem Enterprise Topologie-Graphen $G = (V, E)$ handelt es sich um einen gerichteten Graphen, mit der Möglichkeit Mehrfachkanten zu definieren. Um eine eindeutige Semantik und Wiederverwendbarkeit der verwendeten Knoten- und Kantenmengen zu gewährleisten, sind diese typisiert. Die entstandenen Typen können dabei eine beliebig komplexe Vererbungshierarchie bilden.

Ein Enterprise Topologie Graph stellt ein Abbild einer bestehenden Infrastruktur dar. Um die Provisionierung einer solchen strukturierten Infrastruktur in der Cloud entwerfen zu können, werden sogenannte „Provisionierungs-Topologie-Graphen“ (PTG) eingeführt. Durch diese können alle Komponenten und Relationen beschrieben werden, die für eine Provisionierung von Anwendungen, Infrastruktur und Plattformen notwendig sind. Nach der Provisionierung kann die geschaffene Infrastruktur dann durch ETGs dargestellt werden. Die Standardisierung der Modellierung von Provisionierungs-Topologie-Graphen erfolgte durch die Einführung des OASIS-Standards TOSCA [1], welcher in Kapitel 2.3 beschrieben wird.

Da sich diese Diplomarbeit mit der Provisionierung von Anwendungen beschäftigt, wird im Folgenden nicht zwischen ETG und PTG unterschieden. Dabei werden die Begriffe Topologie, Anwendungstopologie und Topologie Graph als Synonym des Begriffs „Provisionierungs-Topologie-Graph“ verwendet.

2.3. TOSCA - Topology and Orchestration Specification for Cloud Applications

In diesem Kapitel werden der Aufbau und die Konzepte des TOSCA-Standards beschrieben. Aufgrund des großen Umfangs werden in diesem Abschnitt lediglich die für diese Arbeit relevanten Inhalte erläutert. Detaillierte Informationen befinden sich in der TOSCA-Spezifikation [4], im TOSCA-Primer [5], sowie in [2], [7], [6] und [11], auf denen dieses Grundlagenkapitel basiert.

TOSCA ist ein XML-basierter Standard der „Organization for the Advancement of Structured Information Standards" (OASIS) [12], welcher Anwendungstopologien und deren Management portabel beschreibt. Um eine Anwendung automatisiert in einer Cloud provisionieren zu können, werden Topologien entworfen, die alle Komponenten und Relationen beschreiben, die für eine Provisionierung erstellt oder benötigt werden. Dabei handelt es sich um gerichtete Graphen, bestehend aus definierten Knoten- und Kantenmengen. Derartige Topologien werden in der TOSCA-Terminologie als „Topology Templates" bezeichnet. Vor der Einführung des TOSCA-Standards wurden Topologien in verschiedenen Formaten sowohl formell als auch grafisch angefertigt (siehe bspw. [13]). Die Standardisierung der Topologie-Modellierung bringt jedoch diverse Vorteile mit sich. Darunter Portierbarkeit und somit ein Verhindern von Vendor-Lock-ins [6] sowie Kosten- und Aufwandsersparnis für Cloud-Provider und Service-Entwickler. Die Komponenten einer Topologie sind dabei beispielsweise die Anwendung selbst (mitsamt allen Teil-Komponenten), Web Server, Betriebssysteme oder Datenbanken. Als „Relationen" bezeichnet man die Verbindungen zwischen diesen Komponenten (z.B. „connected to", „hosted on"). In der TOSCA-Terminologie werden diese Komponenten als „Node Templates", die Relationen als „Relationship Templates" bezeichnet. Diese Templates werden von vordefinierten Typen („Node Type", „Relationship Type") abgeleitet, die diverse Eigenschaften und Restriktionen der Templates vorgeben (siehe Abbildung 2.3). Die Verwendung von Typen sorgt dabei für Wiederverwendbarkeit und somit Aufwandsersparnis bei der Modellierung. Zusätzlich wird dadurch die Semantik der Elemente eindeutig definiert. Die von den Typen vorgegebenen Eigenschaften („Property Definitions") müssen bei der Instanziierung eines Templates durch konkrete Werte befüllt werden. Hierbei können bei der Typ-Definition auch Default-Werte der Eigenschaften angegeben werden. Jedes modellierte Node Template besitzt darüber hinaus eine Schnittstelle, über die verschiedene Operationen angeboten werden können. Typischerweise führen diese Operationen administrative Aufgaben wie die Installation, das Starten oder Stoppen der Komponente durch. Des Weiteren existieren in TOSCA sogenannte Artefakte, die ausführbare Dateien wie Skripte oder Kompilate beschreiben, die derartige Operationen ausführen. Diese können analog zu Node und Relationship Templates als Artifact Templates beschrieben werden. Dabei werden auch Artifact Templates von vordefinierten Typen (Artifact Types) abgeleitet. Um den Ablauf der Provisionierung und den definierten Operationen zu beschreiben werden Pläne definiert. Dabei handelt es sich um Workflows, die in einer beliebigen Workflow-Beschreibungssprache, wie z.B. BPEL [14], definiert wurden. Ein Build-Plan gibt beispielsweise vor, welche Operationen der Templates in welcher Reihenfolge aufgerufen werden müssen, um eine Anwendung zu provisionieren. Für das Beschreiben nicht-funktionaler

Anforderungen an Komponenten und Relationen bietet TOSCA die Möglichkeit Policys in einer beliebigen Policy-Beschreibungssprache (z.B. WS-Policy [14]) zu definieren [15].

Symbole der grafischen TOSCA-Darstellung

Um die Grundlagen und Konzepte zu vereinfachen wird im Rahmen dieser Arbeit neben der XML-Darstellung eine grafische TOSCA-Darstellung eingeführt. In diesem Abschnitt werden die Symbole beschrieben, die bei der grafischen Modellierung von TOSCA-Topologien genutzt werden.

Die folgenden Symbole dienen vor allem der Darstellung der Grundlagen. Für die grafische Modellierung von Beispiel-Topologien wird hingegen die visuelle TOSCA-Notation Vino4TOSCA[16] genutzt.

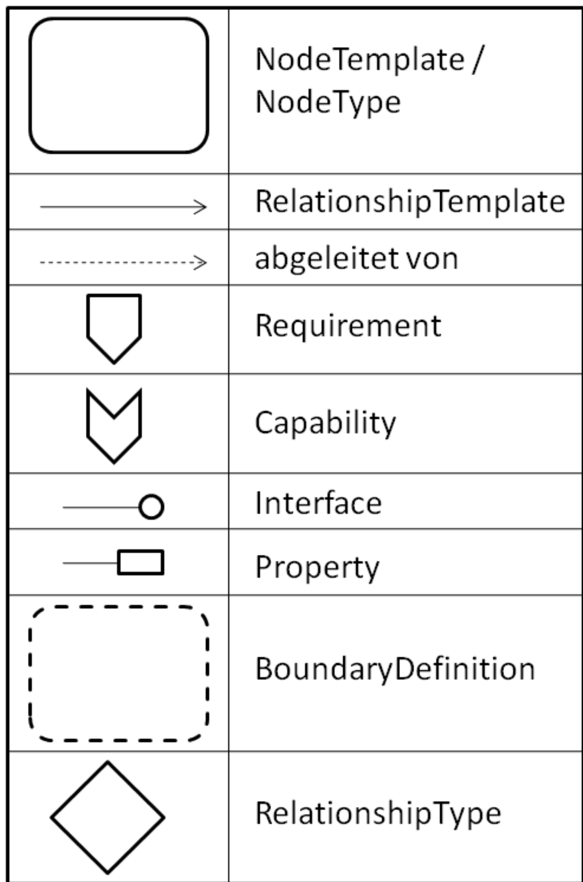

Abbildung 2.2.: Grafische TOSCA-Notation (aus [4] bzw. [5])

Der allgemeine Aufbau eines TOSCA-Topology Templates wird in Abbildung 2.3 mittels grafischer TOSCA-Modellierung verdeutlicht:

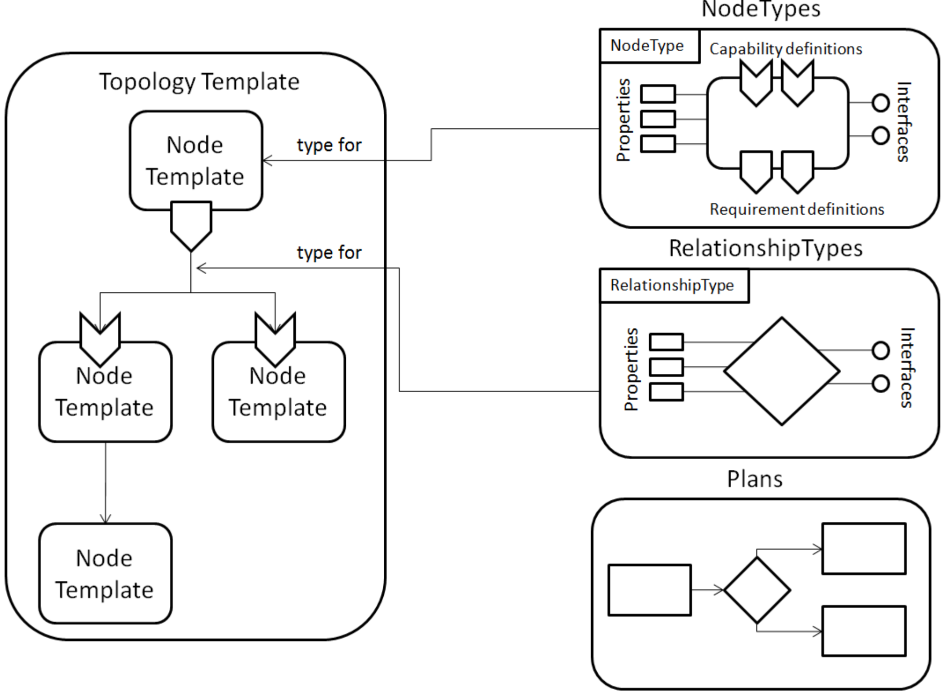

Abbildung 2.3.: TOSCA Überblick (angelehnt an [4])

In TOSCA ist es möglich eine beliebig komplexe Vererbungshierarchie von definierten Node und Relationship Types zu schaffen. Das Vererbungsprinzip wird in Abbildung 2.4 für Node Types verdeutlicht (für Relationship- und Artifact Types analog):

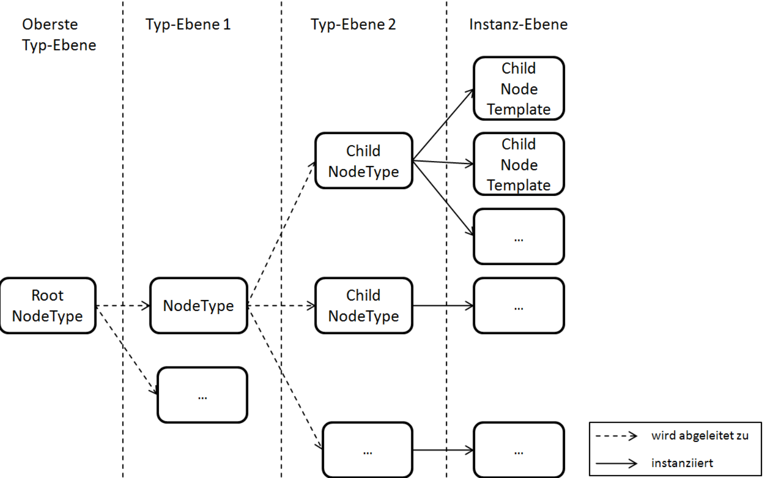

Abbildung 2.4.: Vererbung von TOSCA-Typen

Im Folgenden wird der Aufbau eines TOSCA-Definitions-Dokuments beschrieben. Ein TOSCA-Definitions-Dokument ist eine, in verschiedene Abschnitte untergliederte, XML-Datei, die sowohl die modellierte Topologie (Topology Template mit allen definierten Node und Relationship Templates) enthält, als auch die Definition von Typen, Plänen und Artefakten. In Tabelle 2.1 werden die einzelnen Elemente des TOSCA-Definitions-Dokuments genauer beschrieben.

Tabelle 2.1.: Aufbau eines TOSCA-Definitions-Dokuments

Import	Ermöglicht es Typ-Definitionen aus weiteren TOSCA-Dokumenten einzubinden. Die Verwendung von extern definierten Typen hält den Umfang des Dokuments gering und dieses dadurch übersichtlich.
Types	In diesem Abschnitt können eigene, innerhalb der XML-Datei verwendete, Typen mittels XML-Schema definiert werden. Diese Typen dienen der Definition komplexer Typ-Eigenschaften (bspw. von Node und Relationship Types).
NodeType	Verwendete Node Types mitsamt ihren Eigenschaften, Zuständen, Schnittstellen und Operationen.
RelationshipType	Verwendete Relationship Types mitsamt ihren Eigenschaften, Zuständen, Schnittstellen und Operationen. Ein Relationship Type enthält die optionalen Unterelemente ValidSource und ValidTarget, an denen festgelegt wird, mit welchen Typen von Node Templates dieser verbunden werden kann.
ArtifactType	Beschreibt Typ und Struktur der Artefakte.
ServiceTemplate	Ein Service Template ist das Ober-Element eines Topology Templates. Es enthält neben der Topologie Policy- und Schnittstellen-Informationen sowie Boundary-Definitions (siehe 2.3.3).
TopologyTemplate	Die Topologie, die alle modellierten Templates basierend auf den definierten Typen enthält.
NodeTemplate	Node Template-Definitionen auf Basis der Node Types.
RelationshipTemplate	Relationship Template-Definitionen auf Basis der Relationship Types.
ArtifactTemplate	Beschreibt, für das Deployment benötigte, Artefakte (Skripte / Ausführbare Dateien).
Plans	Definition von Plänen mittels beliebigen Workflow-Beschreibungssprachen wie BPEL. Meist wird an dieser Stelle ein Build-Plan definiert, der beschreibt welche Operationen ausgeführt werden müssen, um die Anwendung zu provisionieren oder zu managen.

Im folgenden Listing 2.1 ist der Aufbau des Definitions-Dokuments als XML dargestellt:

```
<xml version="1.0">
  <Definitions name="" id="" targetNamespace="">

    <Import/>

    <Types/>

    <NodeType name="">
      <documentation/>
      <NodeTypeProperties element=""/>
      <InstanceStates/>
      <Interfaces>
        <Interface name="">
          <Operation name="">
            <InputParameters/>
          </Operation>
        </Interface>
      </Interfaces>
    </NodeType>

    <RelationshipType name="">
      <documentation/>
      <ValidSource typeRef=""/>
      <ValidTarget typeRef=""/>
    </RelationshipType>

    <ArtifactType name="" targetNamespace=""/>

    <ServiceTemplate id="">
      <TopologyTemplate id="">
        <NodeTemplate id="" name="" type=""/>
        <RelationshipTemplate id="" name="" type="">
          <SourceElement ref=""/>
          <TargetElement ref=""/>
        </RelationshipTemplate>
      </TopologyTemplate>
    </ServiceTemplate>

    <ArtifactTemplate id="" name="" type=""/>

    <Plans/>

  </Definitions>
</xml>
```

Listing 2.1: TOSCA-XML-Aufbau

Abbildung 2.5 zeigt ein Beispiel einer mit Vino4TOSCA[16] grafisch modellierten TOSCA-Topologie. Diese wurde mit dem TOSCA-Modellierungstool Winery erstellt. Der zugehörige TOSCA-XML-Code befindet sich in Anhang A.1.
Hierbei wurde eine Topologie für die Provisionierung einer, mit einer PostgreSQL-Datenbank verbundenen, PHP-Anwendung in der Amazon-Cloud mittels Node und Relationship Templates beschrieben.

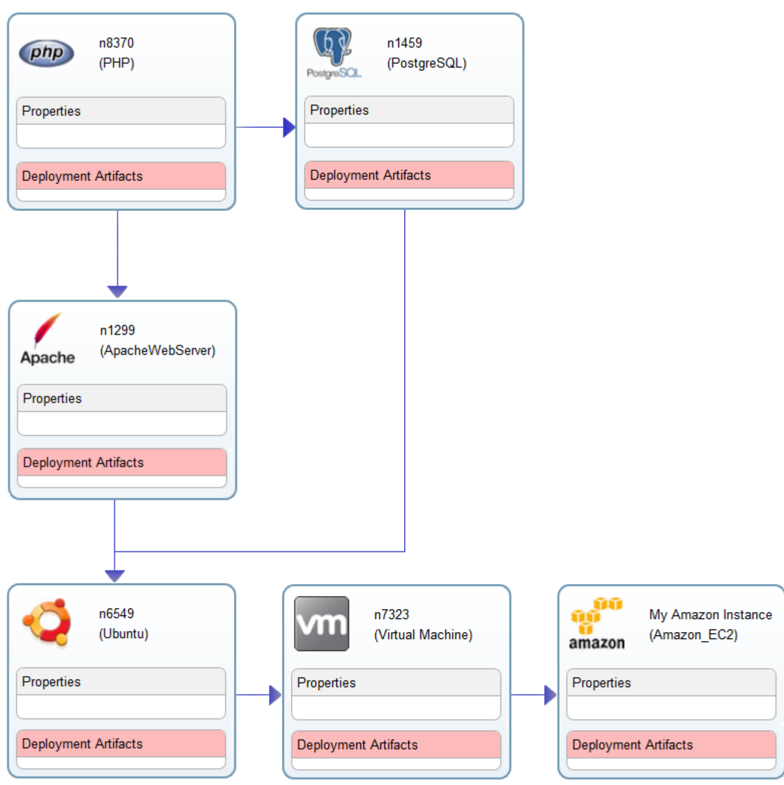

Abbildung 2.5.: Grafische TOSCA Beispiel-Topologie

2.3.1. Cloud Service Archive (CSAR)

Die mittels XML beschriebenen Typen, Pläne, Artefakte und Topology Templates werden in einem, als „Cloud Service Archive“ bezeichneten, Archiv zusammengefasst. Dieses enthält neben den XML-Definitionen Informationen über Metadaten sowie definierte Artefakte als ausführbare Dateien. Das CSAR enthält also alle Elemente, die für eine Provisionierung und das Management einer Anwendung erforderlich sind. Es ist ein in sich geschlossenes (engl.: self-contained) Archiv. Bei einem CSAR handelt es sich typischerweise um eine Datei im ZIP-Format. Das genaue Format ist in der TOSCA-Spezifikation jedoch nicht vorgegeben. Im Folgenden werden die einzelnen Inhalte des CSAR beschrieben.

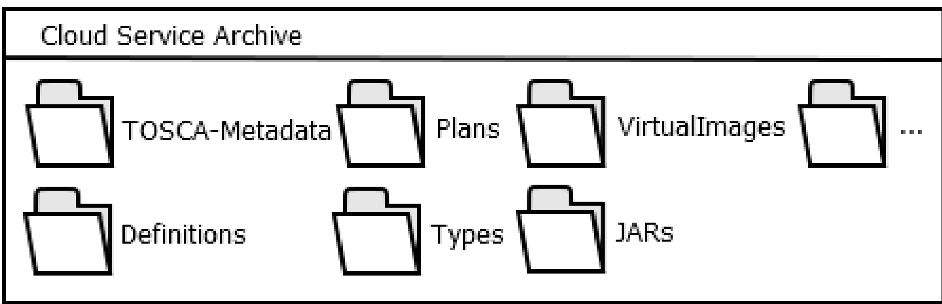

Abbildung 2.6.: Ordnerstruktur CSAR [4]

- **TOSCA-Metadata**
  Jedes CSAR enthält die Datei TOSCA.meta, in der die Meta-Daten beschrieben werden. Die Meta-Datei ist eine einfache Textdatei, die in, durch Leerzeilen getrennte, Blöcke unterteilt ist. Jeder dieser Blöcke beschreibt eines der im CSAR enthaltenen Elemente (bspw. Artefakte). Dabei werden jeweils Name-Wert-Paare für Name und Typ des Inhalts angegeben.
- **Definitions**
  Enthält das Definitions-Dokument sowie alle weiteren, per XML beschriebenen, Service Templates und Typen.
- **Artifacts**
  Enthält die verwendeten Artefakte als ausführbare Dateien oder Skripte.
- **Plans**
  Enthält Pläne (Workflows), die in einer Workflow-Beschreibungssprache definiert wurden.
- **Types**
  Enthält vordefinierte Typen, die mittels Import in das Definitions-Dokument integriert werden.

In den nachfolgenden Unterkapiteln werden spezielle TOSCA-Features beschrieben, die im Rahmen dieser Arbeit verwendet werden.

2.3.2. Requirements und Capabilities

TOSCA ermöglicht es Requirements (dt.: Voraussetzungen) an TOSCA-Node Templates anzufügen. Dadurch kann vorgegeben werden, dass ein Node Template nur dann mit einem weiteren verbunden werden kann, wenn die durch das Requirement festgelegten Bedingungen durch dieses erfüllt werden. Umgekehrt ist es möglich, die Fähigkeiten einer Komponente (Capabilitys) zu definieren. Um eine vollständige TOSCA-Topologie zu erhalten, muss jedes Requirement durch mindestens eine Capability erfüllt sein. Durch welche Relationship Templates mit Requirements und Capabilitys versehene Node Templates verbunden werden können, wird durch die Attribute „ValidSource" bzw. „ValidTarget" der Relationship Types festgelegt. Requirements und Capabilitys können dabei auch an Node Types angefügt werden (TOSCA: Requirement/CapabilityDefinitions). Sobald ein weiterer Typ von diesem abgeleitet wird, muss auch dieser die Requirements und Capabilitys des Obertyps enthalten.
Analog zu Node und Relationship Templates werden auch Requirements und Capabilitys von vordefinierten Typen abgeleitet. Dies wird in folgender Abbildung verdeutlicht:

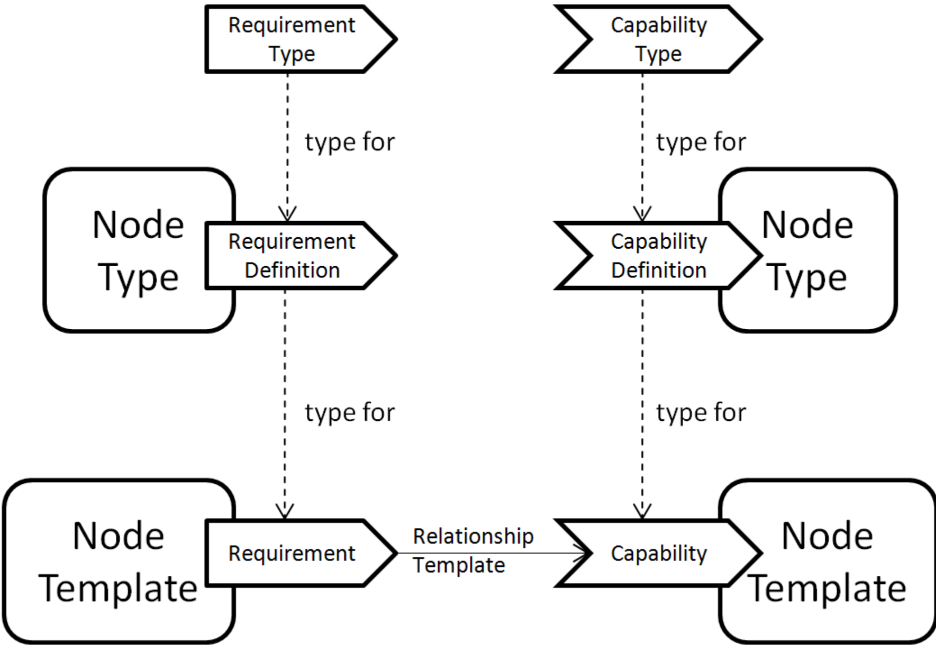

Abbildung 2.7.: TOSCA Requirements und Capabilitys (angelehnt an [4])

2.3.2.1 Anwendungsbeispiel: Verwendung von Requirements

Da Requirements und Capabilitys für diese Arbeit essentiell sind, werden diese anhand eines Anwendungsszenarios genauer beschrieben. Wie bereits erwähnt, legt der Modellierer technische Anforderungen an Node Templates mittels Requirements fest. Dabei kann beispielsweise angegeben werden, auf welchen Komponenten eine Anwendung deployt werden soll oder ob weitere Komponenten benötigt werden, um eine vollständige Topologie zu erhalten. Das folgende Beispiel zeigt eine typische Verwendung von Requirements. Das PHP-Node Template legt dabei durch die Requirements „PHPContainerRequired" und „MySQLDatabaseRequired" fest, dass es sowohl einen Container benötigt auf dem es deployt werden kann, als auch eine MySQL-Datenbank zur Datenspeicherung. Eine Verbindung ist somit nur zu Node Templates möglich, die diese Requirements erfüllen. Nach dem Erfüllen der Requirements durch Node Templates mit passender Capability ist zu sehen, dass diese über ihre Typen wiederum Requirements enthalten können, die erfüllt werden müssen. In dieser Beispiel-Topologie fordert der Web-Server ein Host-Betriebssystem, die Datenbank ein Datenbank-Management-System.

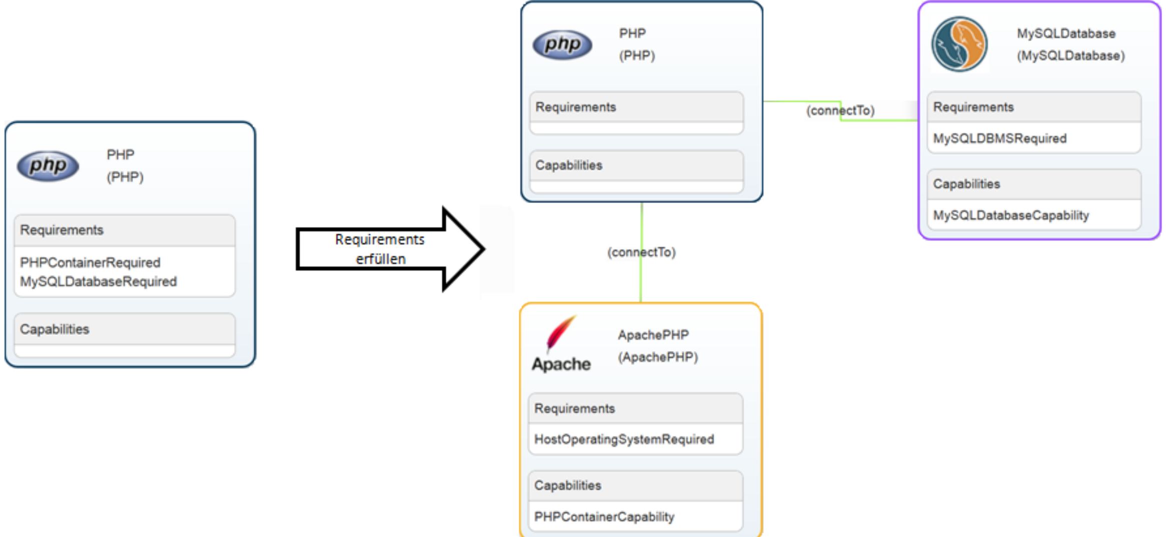

Abbildung 2.8.: Beispiel für die Verwendung von Requirements

2.3.2.2 Matching von Requirement und Capability

Um eine zu einem Requirement passende Capability eines Node Templates bzw. Types zu finden, muss der Typ des Requirements betrachtet werden. Dieser enthält das XML-Attribut *requiredCapabilityType*, welches den Typ der passenden Capability enthält. Alle Node Templates die eine Capability dieses Typs besitzen, können mit dem Requirement verbunden werden.

2.3.3. Node Template-Ersetzung

TOSCA bietet die Möglichkeit einzelne Node Templates durch Service Templates zu ersetzen. Um eine Ersetzung möglich zu machen müssen sogenannte Boundary Definitions angegeben werden, die durch das ersetzende Service Template erfüllt werden müssen. Bei den Boundary Definitions handelt es sich um TOSCA-Properties, Requirements oder Capabilities, die bei dem einzufügenden Service Template übereinstimmen müssen. Ist dies der Fall, kann ein einzelnes Node Template durch ein vollständiges Service Template mit einer beliebigen Anzahl an enthaltenen Node und Relationship Templates ersetzt werden.

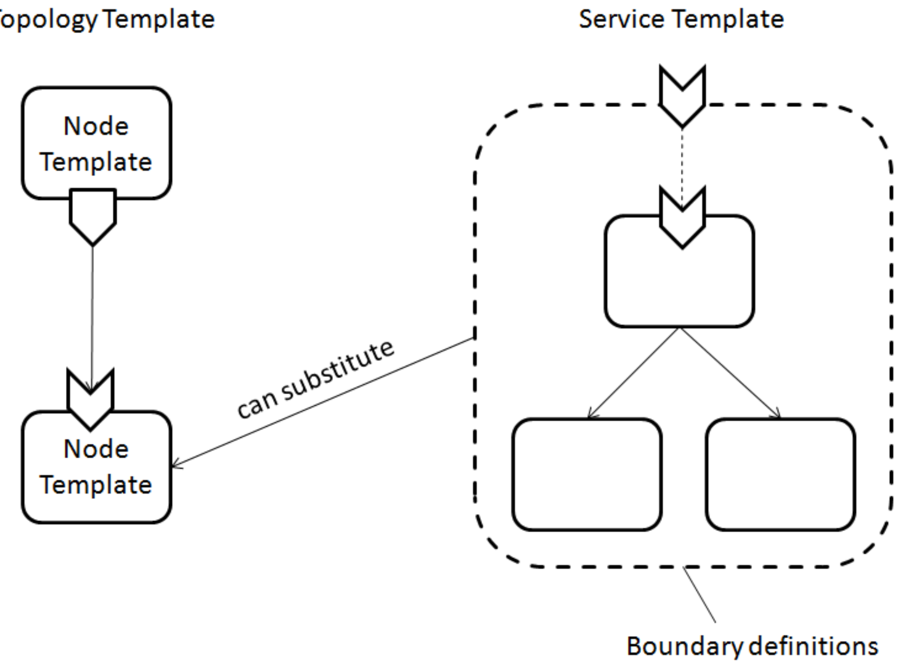

Abbildung 2.9.: Ersetzung von Node Templates (Node Template Substitution) [4]

2.3.4. Abstrakte Typen

Ein TOSCA-Node oder Relationship Type kann durch Setzen des XML-Attributs *abstract* als abstrakt definiert werden. Aus abstrakten Typen können keine Node bzw. Relationship Templates instanziiert werden. Vergleichbar ist dieses Prinzip mit abstrakten Klassen aus diversen Programmiersprachen, aus denen keine Objekte erzeugt werden können. Um Templates aus einem abstrakten Typ zu instanziieren, muss zuerst ein konkreter Typ von dem abstrakten Node Type abgeleitet werden, der anschließend zur Instanziierung der Templates dient. Diese Vorgehensweise ist dann sinnvoll, wenn durch die abstrakten Typen allgemeine Eigenschaften vorgegeben werden, die von mehreren abgeleiteten Node Types erfüllt werden müssen. Somit kann eine Mehrfach-Definition dieser Eigenschaften umgangen werden. Beispielsweise ist es sinnvoll einen allgemeinen, abstrakten Typ „Betriebssystem“ einzuführen, von dem die konkreten Typen „Windows“, „Ubuntu“, „RedHat“ etc. ableiten. Im abstrakten Obertyp können dabei Eigenschaften, sowie Requirements und Capabilitys definiert werden, die alle abgeleiteten Typen enthalten müssen.
Abstrakte Typen werden im Rahmen der Vervollständigung bei der Definition von Platzhaltern (siehe Kapitel 4.3.3) verwendet.

2.3.5. TOSCA und Policys

Mit Hilfe einer Policy-Beschreibungssprache (z.B. WS-Policy [14]) können in TOSCA nicht-funktionale Anforderungen an eine Topologie definiert und anschließend für die Provisionierung und das Management verwendet werden. Typische nicht-funktionale Anforderungen sind bspw. die Verfügbarkeit, Skalierbarkeit oder der Speicher-Standort von Daten. Die genaue Ausprägung der Policys ist abhängig von der genutzten Beschreibungssprache. Die Policy wird in das TOSCA-Definitions-Dokument als Unterelement eines Policy Templates übernommen. TOSCA bietet dabei die Möglichkeit Policy Types zu definieren, um die Wiederverwendbarkeit definierter Policys zu gewährleisten. Eine beliebige Anzahl definierter Policy-Templates kann dabei an Node oder Relationship Templates über das XML-Unterelement „Policies“ angefügt werden [15]. Auch im Rahmen der Diplomarbeit können Policys für die nicht-funktionalen Anforderungen verwendet werden. Dabei wird jedoch keine der etablierten Policy-Beschreibungssprachen genutzt, sondern ein eigenes Policy-Format festgelegt (siehe Ausblick-Kapitel 7.1).

2.3.6. Winery - Ein grafisches TOSCA-Modellierungstool

Neben der XML-Darstellung wurde vom Institut für Architektur von Anwendungssystemen [17] eine visuelle Notation für TOSCA (Vino4TOSCA)[3] [16] geschaffen. Diese kann mit Hilfe des webbasierten Modellierungstools Winery [2] (Teil des OpenTOSCA-Projektes [7]) modelliert werden, welches clientseitig in Javascript und serverseitig in Java implementiert ist. Die toolunterstützte TOSCA-Modellierung ermöglicht es, nicht mehr direkt auf XML-Code-Ebene modellieren zu müssen. Dies fördert Einfachheit, Kommunikation und Erlernbarkeit von TOSCA-Topologien. Da sich Winery während dieser Arbeit noch in der Entwicklung befindet, sind noch nicht alle im TOSCA-Standard genannten Inhalte grafisch modellierbar. Durch die Möglichkeit den XML-Code einer Topologie auch direkt zu bearbeiten sind diese dennoch umsetzbar. Die folgenden Screenshots zeigen den Modellierungs-Editor von Winery, sowie die Oberfläche des Repositorys mit integriertem XML-Editor.

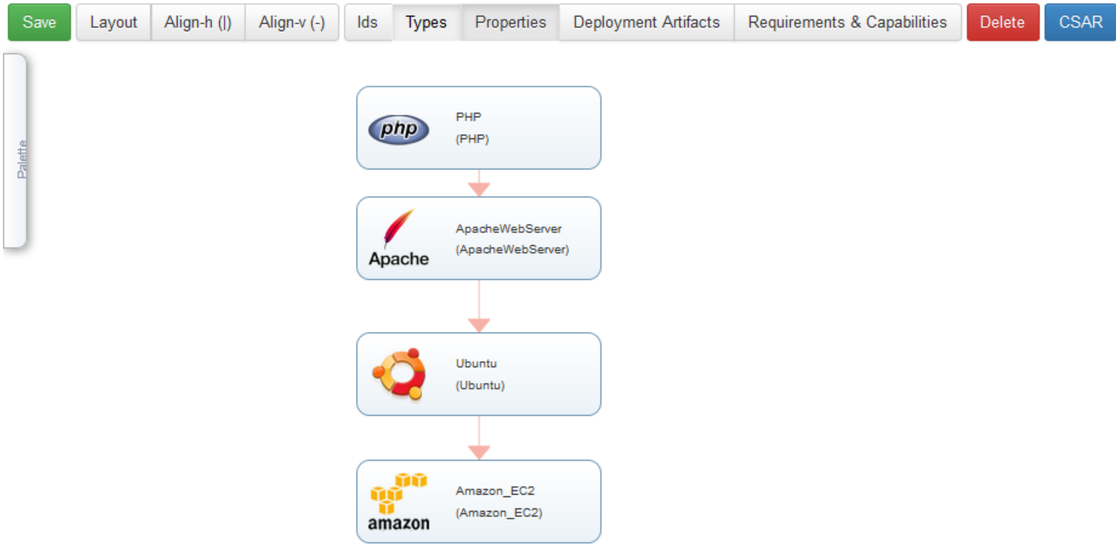

Abbildung 2.10.: Topologie-Modellierung in Winery

Wie in Abbildung 2.10 zu sehen ist, bietet der Topologie-Modellierer von Winery die Möglichkeit eine Topologie grafisch nach Vino4TOSCA anzufertigen. Hierfür können im linken Bildschirmbereich Node Types aus der Palette ausgewählt und per Drag & Drop in den Editier-Bereich verschoben werden. Dabei werden Node Templates aus den Typen instanziert und in die Topologie eingefügt. Nachdem die Node Templates in den Editierbereich eingefügt wurden, können die Verbindungen (Relationship Templates) zwischen diesen - ebenfalls per Drag & Drop - erstellt werden. Requirements, Capabilitys und Artefakte können anschließend über die Buttons der Werkzeug-Leiste an die Node Templates angefügt

[3] www.vino4tosca.org

werden. Im Hintergrund wird dabei eine Topologie im XML-Format geschaffen, die anschließend exportiert werden kann. Der Topologie-Modellierer von Winery bietet darüber hinaus Funktionen, um eine Topologie zu layouten, abzuspeichern oder zu löschen.

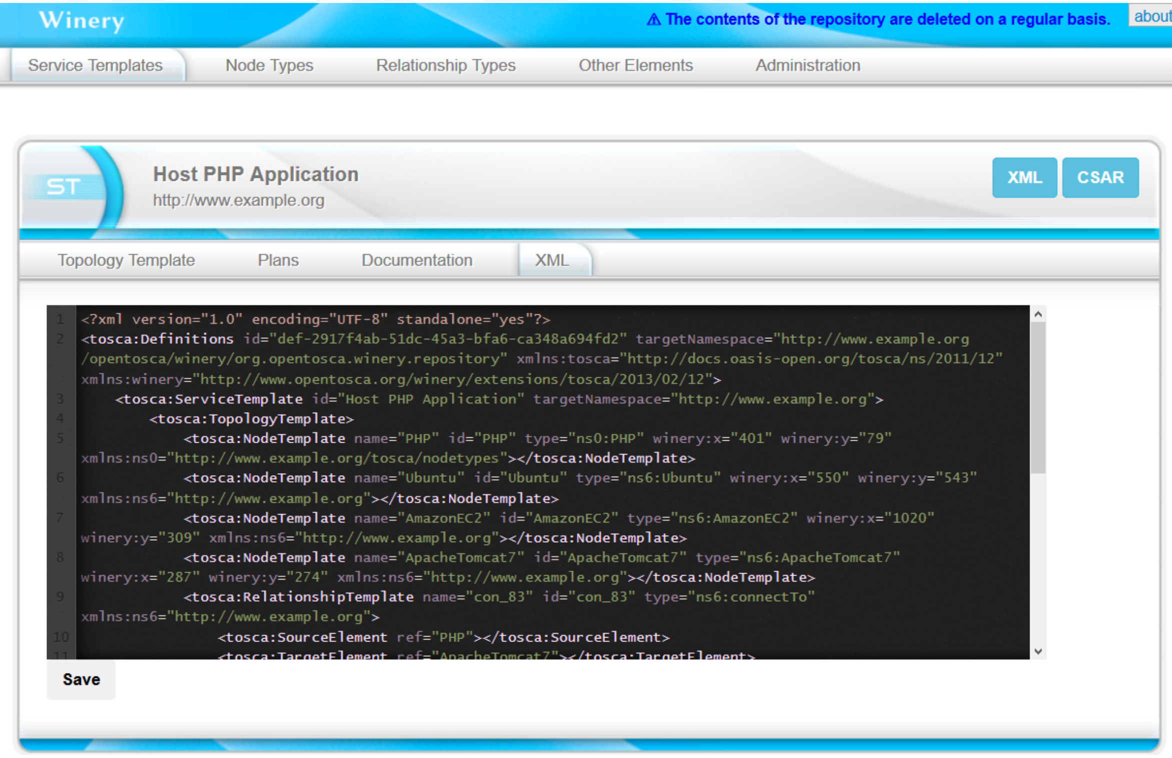

Abbildung 2.11.: Oberfläche des Winery Repositorys

Abbildung 2.11 zeigt die Oberfläche des Winery-Repositorys, welches der Verwaltung von Topologien und den verwendeten Typen dient. In diesem können Topologien, Node und Relationship Types, Artefakte, Policys und weitere TOSCA-Elemente angelegt, editiert und gelöscht werden. Das Repository stellt also die Basis der Topologie-Modellierung dar. Der integrierte XML-Editor macht es neben der grafischen Modellierung möglich, eine Topologie auch auf XML-Ebene zu bearbeiten. Die Änderungen werden dabei automatisch in die grafisch modellierte Topologie übernommen. Nach der Modellierung kann der Export genutzt werden, um ein CSAR aus der Topologie zu erstellen.

3. Problembeschreibung - Vervollständigung von TOSCA-Topologien

Nachdem die Aufgabenstellung dieser Diplomarbeit in den vorigen Kapiteln bereits grob beschrieben wurde, wird sie in diesem Abschnitt weiter verfeinert. Dabei wird auch auf Herausforderungen, die im Verlauf der Arbeit zu bewältigen sind, eingegangen.

3.1. Aufgabenstellung

Die Aufgabenstellung ist in zwei Teile gegliedert:

1. Modellierung unvollständiger TOSCA-Topologien

Unter einer unvollständigen TOSCA-Topologie versteht man ein TOSCA Topology Template, welches lediglich anwendungsspezifische Komponenten, wie zum Beispiel Datenbanken oder Skripte, enthält und die Modellierung von Infrastruktur- und Plattform-Komponenten offen lässt. Durch die Anfertigung unvollständiger TOSCA-Topologien soll dem Modellierer möglichst viel Aufwand erspart werden. Besonders die Modellierung von Infrastruktur-Komponenten erfordert viel Zeit, Aufwand und Know-how des Modellierers. Beispielsweise müssen bei der Modellierung eines Web-Servers Details der Server-Konfiguration angegeben werden. In den meisten Fällen haben Service-Entwickler an derartige Infrastruktur-Komponenten jedoch keine konkreten Anforderungen. Ihr Ziel ist es lediglich die Anwendung erfolgreich in der Cloud zu provisionieren.

Die Aufgabe des ersten Teils dieser Diplomarbeit ist es also, eine Möglichkeit auf Basis des TOSCA-Standards zu schaffen, um derartige unvollständige Topologien zu modellieren. Dabei sollen weiterhin valide TOSCA-Topologien entstehen. Für die Umsetzung können die verfügbaren Konzepte und Methoden des TOSCA-Standards genutzt oder erweitert werden. Das zu erarbeitende Konzept soll möglichst alle Anwendungsfälle bei der Modellierung unvollständiger Topologien abdecken.

2. Vervollständigung unvollständiger TOSCA-Topologien

Nachdem der Modellierer eine unvollständige TOSCA-Topologie modelliert hat, soll diese automatisch zu einer provisionierbaren Topologie vervollständigt werden. Dabei müssen passende Infrastruktur-, Plattform- und auch Anwendungs-Komponenten gefunden und per Node Templates zur Topologie hinzugefügt werden. Anschließend sind die neu hinzugefügten Node Templates durch passende Relationship Templates zu verbinden, um eine zusammenhängende Topologie zu erhalten. Durch die Vervollständigung entsteht letztendlich eine Topologie, durch die eine Anwendung automatisiert provisioniert werden kann. Die Voraussetzungen hierfür sind Validität und Vollständigkeit der Topologie. Vollständig bedeutet in diesem Zusammenhang, dass alle notwendigen Komponenten für eine automatisierte Provisionierung in der Topologie enthalten sind. Dabei ist zu beachten, dass manche TOSCA-Umgebungen (bspw. OpenTOSCA) vollständige Topologien benötigen, andere (bspw. IBM SCO) auch unvollständige Topologien verarbeiten können. Je nach gewählter Umgebung kann die Definition der Vollständigkeit daher variieren.

3.2. Herausforderungen

Aus der oben beschriebenen Aufgabenstellung ergeben sich einige Herausforderungen, die im Rahmen des Konzeptes gelöst werden müssen. Auf die Lösungen wird in Kapitel 4 eingegangen.

Schaffen von Freiheiten für den Modellierer
Das Ziel dieser Arbeit ist es dem Topologie-Modellierer möglichst viele Freiheiten bei der Modellierung zu bieten. Dabei soll es einerseits möglich sein keinerlei Einschränkungen an die Topologie-Vervollständigung festzulegen, andererseits auch die Möglichkeit geben konkrete Vorgaben an diese zu stellen, um ein zufriedenstellendes Ergebnis zu erreichen. Die Anforderungen sollen dabei sowohl technischer als auch nicht-technischer Art sein.

Suche passender Komponenten
Die Suche nach Komponenten, mit denen eine unvollständige Topologie vervollständigt werden kann, stellt sich als größte Herausforderung dar. Es gilt dabei festzustellen, welche Komponenten an welcher Stelle einzufügen sind, wie diese verbunden werden müssen und wo Konflikte entstehen können.

Komponenten-Auswahl
Haben sich die einzufügenden Komponenten gefunden, gibt es für jede dieser Komponenten meist mehrere Alternativen. Aus der Menge an Alternativen müssen einzufügende Node Templates bestimmt werden.

Erzielen verwertbarer Ergebnisse
Die erzeugten Topologien müssen zum einen zum TOSCA-XML-Schema konform sein, zum anderen auf einer TOSCA-Laufzeitumgebung provisionierbar sein, um für den Modellierer

als verwertbar eingestuft werden zu können. Dabei können nicht alle Laufzeitumgebungen dieselben Topologien provisionieren. Dies ist abhängig von deren Funktionsumfang.

Festlegen nicht-funktionaler Anforderungen
Neben den funktionalen Anforderungen müssen auch nicht-funktionale Anforderungen beachtet werden. Diese sollten durch den Modellierer per Policy definiert und bei der Vervollständigung berücksichtigt werden.
Anmerkung: Aufgrund des Umfangs konnte dieser Ansatz im Rahmen dieser Arbeit nicht umgesetzt werden. Das Konzept für das Festlegen nicht-funktionaler Anforderungen ist jedoch im Ausblick-Kapitel 7.1 beschrieben.

Umgang mit mehreren Lösungen
In den meisten Fällen können durch die Vervollständigung mehrere Lösungs-Topologien entstehen. Es muss also entschieden werden, welche der Topologien als Lösung verwendet werden soll.

Feststellen der Vollständigkeit
Ob eine Topologie vollständig, also provisionierbar ist, hängt von der jeweiligen TOSCA-Laufzeitumgebung ab. Die Überprüfung auf Vollständigkeit erfordert also auch eine Untersuchung auf Provisionierbarkeit auf der verwendeten Laufzeitumgebung.

Basis einzufügender Typen
Um passende Node Templates einfügen zu können, muss eine Basis vorhandener Node Types existieren, aus denen diese initialisiert werden können. Analog müssen Relationship Types vorhanden sein, mit denen diese verbunden werden. Benötigt wird also ein Repository, welches die Typen für die Vervollständigung enthält. Des Weiteren müssen die Anforderungen, welche über TOSCA-Requirements festgelegt werden können, in diesem Repository abgespeichert werden.

Eigenschaften von Node Templates
Wird ein Node Template instanziiert, besitzt dieses meist mehrere Felder (bspw. Datenbank-Benutzername und -Passwort), die mit konkreten Werten versehen werden müssen. Diese sollten möglichst ohne Nutzer-Interaktion bei der Vervollständigung ergänzt werden.
Anmerkung: Aufgrund des Umfangs konnte auch dieser Ansatz nicht im Rahmen der Arbeit umgesetzt werden. Näheres hierzu im Ausblick-Kapitel 7.2.

4. Konzeptionelle Lösung

In diesem Kapitel werden Konzepte möglicher Lösungsansätze der in Kapitel 3 geschilderten Problemstellung beschrieben. Dabei werden zwei Vorgehensweisen bei der Vervollständigung von Topologien ausgearbeitet. Eine unvollständig modellierte Topologie soll sowohl in einem Schritt, als auch schrittweise, d.h. nutzergeführt vervollständigt werden können. Der schrittweise Ansatz wird dabei als „Assistierte Modellierung“ bezeichnet. Diese beiden Ansätze werden im Folgenden näher beschrieben.

4.1. Vervollständigung einer Topologie in einem Schritt

Bei der Vervollständigung in einem Schritt soll die Nutzerinteraktion während der Vervollständigung weitestgehend minimiert werden. Als Grundlage dient eine unvollständig modellierte TOSCA-Topologie, die so noch nicht auf einer TOSCA-Laufzeitumgebung provisioniert werden kann. Wählt der Modellierer „Vervollständigen“ aus, wird ein Algorithmus angestoßen, der die Topologie durch TOSCA-Elemente (Node und Relationship Templates) ergänzt, bis diese provisionierbar ist. Existieren mehrere mögliche Lösungs-Topologien, werden diese dem Modellierer zur Auswahl gegeben. Eine Interaktion des Modellierers ist also erst nach der abgeschlossenen Vervollständigung - im Falle mehrerer Lösungen - notwendig. Somit kann das Ziel die Nutzer-Interaktion zu minimieren erreicht werden. Hat der Modellierer eine Lösungs-Topologie ausgewählt, kann diese exportiert und für eine automatisierte Provisionierung durch eine TOSCA-Laufzeitumgebung verwendet werden.

4.2. Assistierte Modellierung vollständiger Topologien

Als assistierte Modellierung wird eine schrittweise Vorgehensweise bei der Vervollständigung von TOSCA-Topologien bezeichnet. Im Gegensatz zur Vervollständigung in einem Schritt soll die assistierte Modellierung den Aufwand während der Vervollständigung nicht minimieren, sondern für eine verbesserte Nachvollziehbarkeit sorgen. Hat der Modellierer eine unvollständige, nicht provisierbare TOSCA-Topologie modelliert und wählt daraufhin „Schrittweise Vervollständigen“ aus, wird die assistierte Modellierung angestoßen. Dabei wird dem Modellierer in jedem Schritt der Vervollständigung eine Auswahl einzufügender TOSCA-Elemente gegeben. Hat er eine Auswahl getroffen, werden die ausgewählten Elemente der Topologie hinzugefügt und der nächste Schritt wird eingeleitet. In diesem kann der Modellierer erneut einzufügende TOSCA-Elemente auswählen. Dies wird wiederholt bis alle für eine Provisionierung notwendigen Komponenten in der Topologie enthalten sind.

Durch diesen Ansatz ist es zum einen möglich die Vervollständigung Schritt für Schritt nachvollziehen zu können, zum anderen kann sie nach Belieben durch den Modellierer gesteuert werden. Dies garantiert zufriedenstellende Ergebnisse zu erhalten. Neben der geführten Modellierung wird außerdem die Möglichkeit geschaffen, die schrittweise Vorgehensweise zu unterbrechen, damit der Modellierer jederzeit selbst Änderungen an der Topologie vornehmen kann. Dies ist bei der Ein-Schritt-Vervollständigung nicht möglich.

Um diese beiden Ansätze umsetzen zu können, muss ein Konzept zur Modellierung unvollständiger TOSCA-Topologien entwickelt werden. Bei der Untersuchung dieses Konzeptes haben sich 5 Anwendungsfälle ergeben, die im Folgenden beschrieben werden.

4.3. Anwendungsfallanalyse: Modellierung unvollständiger Topologien

Um dem Nutzer die im vorigen Kapitel beschriebene Vervollständigung von TOSCA-Topologien zu ermöglichen, wird als erstes die Möglichkeit geschaffen, unvollständige, nicht provisionierbare Topologien zu modellieren, die lediglich anwendungsspezifische Details enthalten und konkrete Infrastruktur- und Plattform-Komponenten offen lassen.

Der folgende Konzeptentwurf behandelt dabei die verschiedenen Anwendungsfälle, die bei der Modellierung dieser Topologien auftreten können. Dabei wird vor allem das Konzept der Requirements und Capabilities genutzt, welches im TOSCA-Grundlagen-Kapitel 2.3.2 näher beschrieben ist. Folgende Anwendungsfälle werden in diesem Kapitel beschrieben:

1. Unvollständige TOSCA-Topologien mit Requirements modellieren
2. Unvollständige TOSCA-Topologien ohne Einschränkungen modellieren
3. Teilmodellierung der Infrastruktur
4. Vereinfachte Teilmodellierung der Infrastruktur

Die folgenden Konzepte werden dabei mit der grafischen TOSCA-Notation Vino4TOSCA visualisiert, um die Komplexität zu verringern. Die Beispiel-Topologien aus diesem Kapitel wurden mit Hilfe des TOSCA-Modellierungs-Tools Winery [2] erzeugt.

Die im Folgenden beschriebenen Anwendungsfälle sind alle der selben Struktur folgend aufgebaut. Jeder Anwendungsfall beschreibt ein Ziel, ein Beispiel-Szenario, einen Ansatz, das Vorgehen bei der Modellierung, das Ergebnis sowie eine, sowohl grafisch als auch im XML-Format, modellierte Beispiel-Topologie. Dabei werden die einzelnen Anwendungsfälle zuerst separat beschrieben. In der praktischen Anwendung werden meist Kombinationen der Anwendungsfälle verwendet. Dies wird im letzten Unterkapitel 4.3.5 dargestellt und untersucht.

4.3.1. Anwendungsfall 1: Unvollständige TOSCA-Topologien mit Requirements

Ziel: Der Topologie-Modellierer möchte die Infrastruktur auf der die Anwendung provisioniert werden soll nicht selbst modellieren, hat aber bestimmte Anforderungen an diese, die bei der anschließenden automatischen Vervollständigung eingehalten werden müssen. Hierbei soll es ihm durch die Auswahl der Anforderungen offen stehen, wie stark die zu ergänzende Infrastruktur eingeschränkt wird.

Beispiel-Szenario: Der Topologie-Modellierer möchte eine PHP-Anwendung in der Cloud provisionieren. Dabei hat er die Anforderung, dass die Anwendung auf einer Webserver-Komponente deployt werden muss, um lauffähig zu sein. Konkrete Anforderungen an die Eigenschaften dieser Komponente oder an die restliche Infrastruktur hat der Modellierer jedoch nicht.

Ansatz: Der Modellierer fügt TOSCA-Requirements an Node Templates, um Anforderungen an die Infrastruktur zu definieren. Des Weiteren kann der Fall auftreten, dass die Topologie nicht alle von den Node Types definierten Requirements erfüllt. Die Requirements stammen dabei aus einer bekannten Menge.

Vorgehen bei der Modellierung:

- **Schritt 1:**
  Der Modellierer modelliert alle anwendungsspezifischen Komponenten mittels TOSCA-Node Templates sowie alle Relationen dieser Komponenten mittels Relationship Templates.
- **Schritt 2:**
  Die modellierten Node Templates werden mit, aus einer vordefinierten Menge stammenden, TOSCA-Requirements versehen, um Anforderungen an die Infrastruktur zu definieren.

Anwendungs-Szenario: Der Modellierer wählt diesen Ansatz, wenn er technische Anforderungen an die Infrastruktur stellt, welche durch TOSCA-Requirements abgedeckt werden können. Hierbei kann bei der Modellierung beispielsweise nicht festgelegt werden, dass die Infrastruktur auf verschiedenen existierenden Maschinen provisioniert werden soll. Es können nur die technischen Details der Maschine(n) durch Requirements eingeschränkt werden.

Auf Basis dieses Anwendungsfalls, kann eine erste Definition vollständiger TOSCA-Topologien erfolgen:

Zwischendefinition I: Vollständige TOSCA-Topologie:
Eine TOSCA-Topologie ist genau dann vollständig, wenn alle Requirements der Node Templates und der verwendeten Node Types erfüllt sind.

Als „erfüllen“ eines Requirements wird dabei das Einfügen eines Node Templates mit einer zum Requirement passenden Capability bezeichnet.

Im Folgenden ist eine unvollständige Beispiel-Topologie dieses Anwendungsfalls dargestellt. Dabei wurde durch TOSCA-Requirements festgelegt, dass die modellierte PHP-Anwendung auf einem Apache PHP-Modul deployt werden soll und die verbundene PostgreSQL-Datenbank durch ein Datenbank-Management-System ergänzt werden muss.

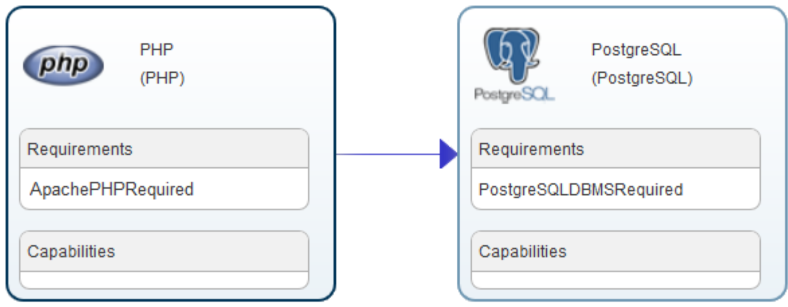

Abbildung 4.1.: Beispiel-Topologie nach Anwendungsfall 1

Der XML-Code der Beispiel-Topologie wird im Folgenden dargestellt:

```
<xml version="1.0">
  <Definitions name="Example_Use_Case_1" id="uc1" targetNamespace="http://www.example.org">
    <ServiceTemplate id="example_service_template">
      <TopologyTemplate id="example_topology_template">
        <NodeTemplate id="PHP" name="PHP" type="PHP">
          <Requirements>
            <Requirement id="ApachePHPRequired" name="ApachePHPRequired"
                         type="ApachePHPRequired"/>
          </Requirements>
        </NodeTemplate>
        <NodeTemplate id="PostgreSQL" name="PostgreSQL" type="Database">
          <Requirements>
            <Requirement id="PostgreSQLDBMSRequired" name="PostgreSQLDBMSRequired"
                         type="PostgreSQLDBMSRequired"/>
          </Requirements>
        </NodeTemplate>
        <RelationshipTemplate id="connectTo" name="connectTo" type="connectTo">
          <SourceElement ref="PHP"/>
          <TargetElement ref="PostgreSQL"/>
        </RelationshipTemplate>
      </TopologyTemplate>
    </ServiceTemplate>
  </Definitions>
</xml>
```

4.3.2. Anwendungsfall 2: Unvollständige TOSCA-Topologien ohne Einschränkungen

Nachdem in Anwendungsfall 1 eine Möglichkeit geschaffen wurde Anforderungen an Topologien festzulegen, beschäftigt sich der zweite Anwendungsfall mit der uneingeschränkten Modellierung unvollständiger TOSCA-Topologien.

Ziel: Der Topologie-Modellierer möchte eine unvollständige TOSCA-Topologie modellieren und dabei **keinerlei** Anforderungen an die zu ergänzende Infrastruktur festlegen. Das Ziel ist lediglich, eine provisionierbare Topologie zu erhalten.

Beispiel-Szenario: Der Modellierer möchte eine PHP-Anwendung in der Cloud provisionieren. Diese besitzt keinerlei Abhängigkeiten zu weiteren Komponenten (Datenbanken etc.). An die Infrastruktur, auf der die PHP-Anwendung provisioniert wird, stellt er keine Anforderungen.

Problematik: Werden anwendungsspezifische Komponenten ohne Requirements modelliert, wird die entstandene Topologie nach Zwischendefinition I als vollständig definiert. Derartige Topologien sind jedoch oftmals nicht provisionierbar, wenn der TOSCA-Laufzeitumgebung weitere Komponenten und Relationen für eine Provisionierung fehlen oder wenn individuelle Node Types verwendet werden, die der Laufzeitumgebung unbekannt sind und daher nicht provisioniert bzw. verarbeitet werden können. Da auch als vollständig definierte Topologien nicht provisionierbar sein können, reicht Definition I nicht aus. Daher wird die Zwischendefinition **„Provisionierbare Topologie"** eingeführt.

Zwischendefinition II: Provisionierbare TOSCA-Topologie:
Eine Topologie ist genau dann provisionierbar, wenn diese alle Komponenten und Relationen enthält, die für die Provisionierung einer Anwendung notwendig sind. Die Provisionierbarkeit einer Topologie ist dabei abhängig von der jeweiligen TOSCA-Laufzeitumgebung.

Anmerkung: TOSCA bietet Modellierern die Möglichkeit individuelle Node Types zu erstellen und für die Instanziierung von Node Templates zu verwenden. Je nach Modellierung dieser Typen kann dies Auswirkungen auf die Provisionierbarkeit haben: falls eine Laufzeitumgebung den modellierten Typ nicht kennt und kein bekanntes Interface implementiert ist (z.B. das TOSCA Lifecycle Interface [5]), kann ein aus diesem Typ instanziiertes Node Template nicht ohne weiteres provisioniert werden, da die TOSCA-Laufzeitumgebung das Node Template weder kennt noch versteht. Da TOSCA keinerlei Node Types als Standard definiert, die alle Laufzeitumgebungen verstehen müssen, ist hier die Portabilität zurzeit noch eingeschränkt. Daher muss in dieser Definition zwischen den verschiedenen Laufzeitumgebungen unterschieden werden.

Annahme: Es existiert eine Prüfung auf Provisionierbarkeit einer Topologie gemäß festgelegten Regeln, wie beispielsweise der Implementierung des TOSCA Lifecycle Interfaces[1].

Vorgehen bei der Modellierung:

- **Schritt 1:**
  Der Modellierer modelliert anwendungsspezifische TOSCA-Node Templates ohne die Verwendung von Requirements und verbindet diese durch Relationship Templates.

Ergebnis: Durch die neu eingeführte Definition kann entschieden werden, ob die Topologie ergänzt werden muss oder ob sie bereits alle für eine Provisionierung notwendigen Komponenten enthält. Somit ist auch eine Modellierung von Topologien ohne Requirements möglich, welche anschließend - abhängig von der TOSCA-Laufzeitumgebung - zu provisionierbaren Topologien ergänzt werden können.

Die folgende Beispiel-Topologie aus Abbildung 4.2 zeigt die Modellierung nach Anwendungsfall 2. Dabei wurde analog zu Abbildung 4.1 eine PHP-Anwendung verbunden mit einer PostgreSQL-Datenbank modelliert. Welcher Webserver und welches Datenbank-Management-System eingefügt werden kann bzw. ob dies überhaupt notwendig ist, wird dabei - in Hinblick auf die TOSCA-Laufzeitumgebung - durch eine eventuelle Vervollständigung und nicht durch den Modellierer festgelegt. Hierbei könnte es beispielsweise der Fall sein, dass eine TOSCA-Laufzeitumgebung diese Topologie bereits ohne Vervollständigung provisionieren kann, eine andere dafür möglicherweise noch zusätzliche Informationen für die Infrastruktur in Form von Node und Relationship Templates benötigt.

Durch die aktuell fehlende Standardisierung von Types und deren Requirements muss, wie beschrieben, zwischen verschiedenen Laufzeitumgebungen unterschieden werden, um die Provisionierbarkeit dieser Topologie feststellen zu können.

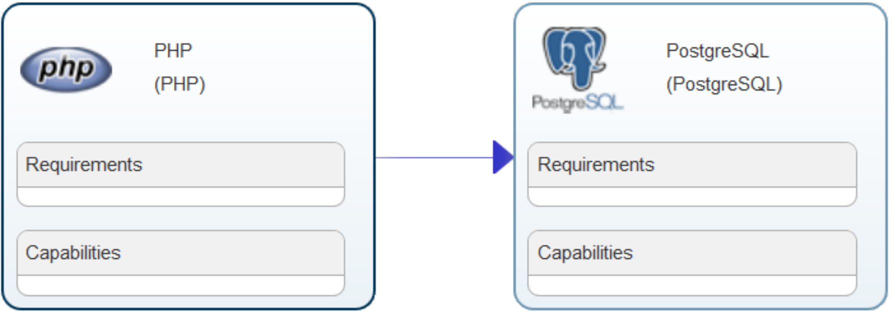

Abbildung 4.2.: Beispiel-Topologie nach Anwendungsfall 2

[1] Diese Annahme kann getroffen werden, da eine Provisionierbarkeits-Prüfung (im Folgenden Provisioning-API genannt) parallel zu dieser Arbeit von Instituts-Seite implementiert wird (siehe Kapitel Umsetzung).

Der XML-Code der Beispiel-Topologie wird im Folgenden dargestellt:

```
<xml version="1.0">
  <Definitions name="Example_Use_Case_2" id="uc2" targetNamespace="http://www.example.org">
    <ServiceTemplate id="example_service_template">
      <TopologyTemplate id="example_topology_template">
        <NodeTemplate id="PHP" name="PHP" type="PHP"/>
        <NodeTemplate id="PostgreSQL" name="PostgreSQL" type="Database"/>
        <RelationshipTemplate id="connectTo" name="connectTo" type="connectTo">
          <SourceElement ref="PHP"/>
          <TargetElement ref="PostgreSQL"/>
        </RelationshipTemplate>
      </TopologyTemplate>
    </ServiceTemplate>
  </Definitions>
</xml>
```

4.3.3. Anwendungsfall 3: Teilmodellierung der Infrastruktur

Ziel: Der Topologie-Modellierer hat bereits konkrete Vorstellungen von Teil-Komponenten der Infrastruktur und möchte diese vollständig modellieren. Die restliche Infrastruktur soll automatisch ergänzt werden.

Beispiel-Szenario: Der Modellierer möchte eine PHP-Anwendung in der Cloud provisionieren und hat diese als vollständig spezifiziertes Node Template modelliert. Dabei besitzt er bereits einen Zugang zum Cloud-Anbieter Amazon und dessen „Elastic Compute Cloud"-Angebot[2]. Um diesen nutzen zu können modelliert er ein Node Template „Amazon EC 2" mit allen notwendigen Details in Form von TOSCA-Properties. An die zwischen Anwendung und Cloud-Anbieter liegenden Komponenten (Webserver, Betriebssystem, Virtuelle Maschine etc.) stellt er jedoch keinerlei Anforderungen. Diese sollen beliebig ergänzt werden. Wichtig ist lediglich, dass sie kompatibel zum modellierten Cloud-Anbieter „Amazon EC 2" sind. Beispielsweise unterstützt nicht jeder Cloud-Anbieter alle Betriebssysteme, daher wird die Auswahl der zu ergänzenden Komponenten hierbei durch die konkrete Modellierung des Cloud-Anbieters eingeschränkt.

Problematik: Es müssen konkrete Teil-Komponenten der Infrastruktur modelliert werden können, die nicht direkt mit den Anwendungskomponenten verbunden sind. Es muss also eine Möglichkeit geschaffen werden, die zwischen der Anwendung und den konkret modellierten Infrastruktur-Komponenten liegenden Node Templates zu abstrahieren.

Ansatz: Führe *typisierte*[3] Platzhalter ein, die Teile der Infrastruktur abstrahieren. Jeder dieser Platzhalter wird dabei vom abstrakten Typ „Placeholder" abgeleitet. Die Typisierung

[2] http://aws.amazon.com/de/ec2/

[3] Die Typisierung ist notwendig, um die Platzhalter voneinander unterscheidbar zu machen. Ein generischer „*-Platzhalter" für alle Node Types ist also in diesem Anwendungsfall nicht möglich.

erlaubt es festzustellen, welchen Zweck der Platzhalter in der Topologie erfüllt (bspw. Datenhaltung, Webserver etc.). Ohne eine Typisierung der Platzhalter müsste dies während der Vervollständigung unter großem Aufwand bestimmt werden, da in diesem Fall prinzipiell alle Node Types für eine Ersetzung infrage kommen würden. Die Typisierung schränkt diese Auswahl ein. Dabei können für Platzhalter keine abstrakten Node Types verwendet werden, da aus diesen keine Node Templates instanziiert werden können. Die Verbindung zu und von den Platzhaltern geschieht durch den generischen Relationship Type „depends on". Dies ist notwendig, da ein Platzhalter durch verschiedene Node Templates ersetzt werden kann, die wiederum unterschiedliche Verbindungen benötigen. Beispielsweise kann ein Platzhalter-Node Template „Webserver" durch die konkreten Node Templates „Apache Tomcat" oder „Glassfish Server" ersetzt werden. Dabei könnte bspw. zum „Apache Tomcat"-Node Template nur mit dem Relationship Type „deployed on Apache" verbunden werden, zu dem „Glassfish Server"-Node Template hingegen nur mit dem Relationship Type „deployed on Glassfish". Daher ist das Verwenden eines generischen Relationship Types für die ein- bzw. ausgehenden Verbindungen der Platzhalter notwendig.

Definition: Platzhalter
Bei einem Platzhalter handelt es sich um ein TOSCA-Node Template, dessen Node Type vom abstrakten Typ „Placeholder" abgeleitet ist. Jeder Platzhalter ist typisiert und dient der Abstraktion konkreter Node Types. Die modellierten Platzhalter müssen durch vollständig spezifizierte Node Templates vom Typ des Platzhalters ersetzt werden, um eine provisionierbare Topologie zu erhalten.

Beispiel: Vererbung von Platzhaltern
Die folgende Abbildung zeigt ein Beispiel der Vererbungshierarchie von Platzhaltern. Hierbei ist zu sehen, dass die Platzhalter „DatabasePlaceholder" und „ApacheWebserverPlaceholder" von Obertypen ableiten, die wiederum vom abstrakten Node Type „Placeholder" abgeleitet sind.

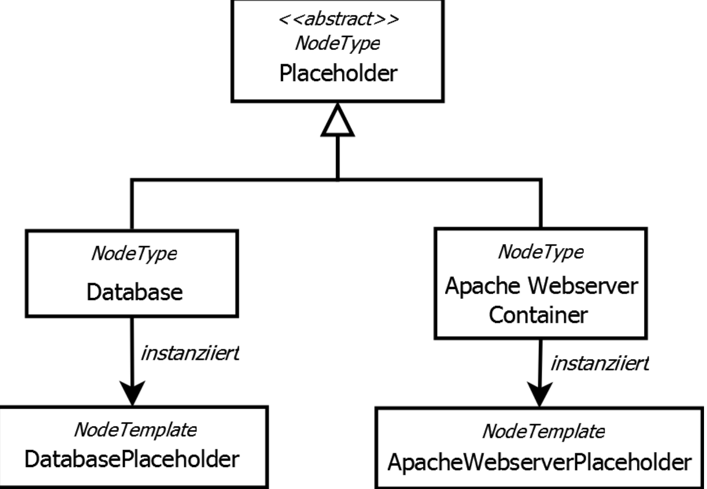

Abbildung 4.3.: Beispiel: Vererbung von Platzhaltern

Vorgehen bei der Modellierung:

- **Schritt 1:**
  Der Modellierer modelliert alle anwendungsspezifischen Komponenten mittels TOSCA-Node Templates sowie alle Relationen dieser Komponenten mittels Relationship Templates.
- **Schritt 2:**
  Der Modellierer verbindet die modellierten Komponenten anschließend mit einer beliebigen Anzahl an Platzhaltern oder konkret modellierten Infrastruktur-Komponenten. Die Verbindung zu den Platzhaltern geschieht dabei über die generische Verbindung „depends on", die im Rahmen der Vervollständigung durch konkrete Relationship Templates ersetzt wird.

Ergebnis: Eine Topologie, die sowohl Anwendungs- als auch Infrastruktur-Komponenten enthält sowie eine beliebige Anzahl an typisierten Platzhaltern.

Durch die Einführung dieses Anwendungsfalls müssen die eingeführten Definitionen wie folgt modifiziert werden:

Zwischendefinition I: Vollständige TOSCA-Topologie:
Eine TOSCA-Topologie ist genau dann vollständig, wenn alle Requirements der Node Templates und der verwendeten Node Types erfüllt sind sowie keine Platzhalter-Node Templates enthalten sind.

Zwischendefinition II: Provisionierbare TOSCA-Topologie:
Eine Topologie ist genau dann provisionierbar, wenn diese keine Platzhalter-Node Templates sowie alle Komponenten und Relationen enthält, die für die Provisionierung einer Anwendung notwendig sind. Die Provisionierbarkeit einer Topologie ist dabei abhängig von der jeweiligen TOSCA-Laufzeitumgebung.

Anzahl und Position der Platzhalter in der Topologie können dabei sehr unterschiedlich ausfallen. Dies wird durch die Beispiel-Topologie aus Abbildung 4.4 verdeutlicht. Dabei wurde durch die Modellierung festgelegt, dass die Anwendung auf einem Ubuntu-Betriebssystem gehostet werden muss, welches in der Amazon Elastic Compute Cloud deployt wird. Der verwendete Webserver wird durch einen Platzhalter abstrahiert und muss während der Vervollständigung durch ein konkretes Webserver-Node Template (bspw. Apache PHP) ersetzt werden. Diese Vorgehensweise ermöglicht es, Infrastruktur-Komponenten einer Topologie entweder genau zu spezifizieren oder offen zu lassen.

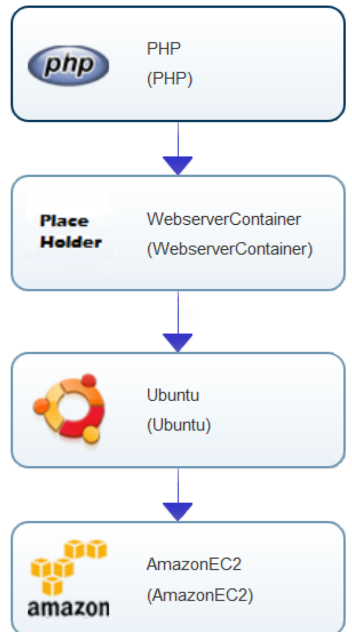

Abbildung 4.4.: Beispiel-Topologie nach Anwendungsfall 3

Der XML-Code der Beispiel-Topologie wird im Folgenden dargestellt:

```
<xml version="1.0">
  <Definitions name="Example_Use_Case_3" id="uc3" targetNamespace="http://www.example.org">
    <NodeType name="WebServerContainer" targetNamespace="http://www.example.org">
      <DerivedFrom typeRef="Placeholder"/>
    </NodeType>
    <ServiceTemplate id="example_service_template">
      <TopologyTemplate id="example_topology_template">
        <NodeTemplate id="PHP" name="PHP" type="PHP"/>
        <NodeTemplate id="WebServerContainer" name="WebServerContainer"
          type="WebServerContainer"/>
        <NodeTemplate id="Ubuntu" name="Ubuntu" type="OperatingSystem"/>
        <NodeTemplate id="AmazonEC2" name="AmazonEC2" type="CloudProvider"/>
        <RelationshipTemplate id="hostedOn" name="hostedOn" type="hostedOn">
          <SourceElement ref="PHP"/><TargetElement ref="WebServerContainer"/>
        </RelationshipTemplate>
        <RelationshipTemplate id="hostedOn" name="hostedOn" type="hostedOn">
          <SourceElement ref="WebServerContainer"/><TargetElement ref="Ubuntu"/>
        </RelationshipTemplate>
        <RelationshipTemplate id="deployOn" name="deployOn" type="deployOn">
          <SourceElement ref="Ubuntu"/><TargetElement ref="AmazonEC2"/>
        </RelationshipTemplate>
      </TopologyTemplate>
    </ServiceTemplate>
  </Definitions>
</xml>
```

4.3.4. Anwendungsfall 4: Vereinfachte Teilmodellierung der Infrastruktur

Ziel: Der Topologie-Modellierer hat Anforderungen an konkrete Teil-Komponenten der Anwendung bzw. der Infrastruktur, möchte jedoch auf die Modellierung von Platzhaltern verzichten, da er keine Anforderungen an Anzahl und Typ der einzufügenden Komponenten hat. Außerdem soll der Aufwand der Modellierung reduziert werden. Dieser Anwendungsfall stellt eine Vereinfachung des dritten Anwendungsfalls dar. In Abschnitt 4.3.4.1 werden diese voneinander abgegrenzt.

Beispiel-Szenario: Der Modellierer möchte eine PHP-Anwendung in der Cloud provisionieren und hat diese als vollständig spezifiziertes Node Template modelliert. Dabei besitzt er bereits einen Zugang zum Cloud-Anbieter Amazon und dessen „Elastic Compute Cloud"-Angebot. Um diesen nutzen zu können modelliert er ein Node Template „Amazon EC 2" mit allen notwendigen Details in Form von TOSCA-Properties. An die zwischen Anwendung und dem konkret modellierten Cloud-Anbieter liegenden Node Templates stellt der Modellierer keine Anforderungen. Dabei sind Typ und Anzahl der eingefügten Komponenten beliebig. Aus diesem Grund sollen keine Platzhalter-Node Templates bei der Modellierung verwendet werden.

Ansatz: Führe einen Relationship Type „Deferred" ein, der als Platzhalter für beliebig viele Node und Relationship Templates dient.
Anmerkung: Der beschriebene Ansatz basiert auf [18].

Vorgehen bei der Modellierung:

- **Schritt 1:**
  Der Modellierer modelliert alle anwendungsspezifischen Komponenten mittels TOSCA-Node Templates sowie alle Relationen dieser Komponenten mittels Relationship Templates.
- **Schritt 2:**
  Der Modellierer modelliert konkrete Anwendungs- oder Infrastruktur-Komponenten durch Node und Relationship Templates und verbindet die bestehende Topologie mittels Relationship Template des Typs „Deferred" zu diesen.

Ergebnis: Eine Topologie, die sowohl Anwendungs- als auch Infrastruktur-Komponenten enthält sowie eine beliebige Anzahl an Relationship Templates des Typs „Deferred". Durch die Einführung des Deferred-Topologie-Anwendungsfalls, müssen die Zwischendefinitionen wie folgt ergänzt werden:

Zwischendefinition I: Vollständige TOSCA-Topologie:
Eine TOSCA-Topologie ist genau dann vollständig, wenn alle Requirements der Node Templates und der verwendeten Node Types erfüllt sind sowie keine Platzhalter-Node Templates und Relationship Templates des Typs „Deferred“ enthalten sind.

Zwischendefinition II: Provisionierbare TOSCA-Topologie:
Eine Topologie ist genau dann provisionierbar, wenn diese weder Platzhalter-Node Templates noch Relationship Templates des Typs „Deferred“ enthält sowie alle Komponenten und Relationen, die für die Provisionierung einer Anwendung notwendig sind. Die Provisionierbarkeit einer Topologie ist dabei abhängig von der jeweiligen TOSCA-Laufzeitumgebung.

Die folgende Grafik verdeutlicht diesen Ansatz, indem das beschriebene Beispiel-Szenario als TOSCA-Topologie modelliert dargestellt ist. Zwischen der modellierten PHP-Anwendung und dem Cloud-Anbieter Amazon können beliebig viele Node und Relationship Templates ergänzt werden, um eine provisionierbare Topologie zu erhalten. Dabei wird es dem Modellierer ermöglicht die PHP-Anwendung sowie die Komponente Amazon EC 2 genau zu spezifizieren, ohne Angaben über die zwischen Anwendung und dieser Komponente liegenden Node und Relationship Templates machen zu müssen.

Abbildung 4.5.: Verwendung von Relationship Templates des Typs „Deferred“

Der XML-Code der Beispiel-Topologie wird im Folgenden dargestellt:

```
<xml version="1.0">
  <Definitions name="Example_Use_Case_5" id="uc5" targetNamespace="http://www.example.org">
    <ServiceTemplate id="example_service_template">
      <TopologyTemplate id="example_topology_template">
        <NodeTemplate id="PHP" name="PHP" type="PHP"/>
        <NodeTemplate id="AmazonEC2" name="AmazonEC2" type="CloudProvider"/>
        <RelationshipTemplate id="deferred" name="deferred" type="deferred">
          <SourceElement ref="PHP"/><TargetElement ref="AmazonEC2"/>
        </RelationshipTemplate>
      </TopologyTemplate>
    </ServiceTemplate>
  </Definitions>
</xml>
```

4.3.4.1 Abgrenzung zu Anwendungsfall 3 - Platzhalter-Modellierung

Auf den ersten Blick scheinen die Anwendungsfälle Deferred-Topologie und Platzhalter-Topologie denselben Zweck zu erfüllen. Jedoch existieren Unterschiede zwischen diesen Ansätzen aufgrund derer beide Anwendungsfälle beibehalten werden: Durch die Modellierung von Platzhaltern kann im Gegensatz zur Deferred-Verbindung festgelegt werden, wie viele Node Templates durch die Vervollständigung eingefügt werden müssen. Außerdem kann die Auswahl der einzufügenden Komponenten durch die Verwendung von Platzhaltern eingeschränkt werden. Beispielsweise kann durch einen Platzhalter „Apache Tomcat Webserver" bereits der Typ der einzufügenden Komponente festgelegt werden. Bei einem Relationship Template des Typs „Deferred" könnten hingegen - bei der Vervollständigung in einem Schritt - beliebige Webserver eingefügt werden.

Ein großer Vorteil bei der Verwendung von Relationship Templates des Typs „Deferred" ist außerdem die erhöhte Flexibilität: Da zum Beispiel verschiedene Cloud Provider unterschiedliche Möglichkeiten bereitstellen, um eine Anwendung zu provisionieren (beispielsweise verschiedene Betriebssysteme), kann die gesamte Topologie durch das Austauschen derartiger Komponenten beeinflusst werden. Wird bspw. der Cloud-Provider ausgetauscht, müssten bei der konkreten Modellierung aller Node Templates ganze Teile der Topologie angepasst werden. Durch die Verwendung von Relationship Templates des Typs „Deferred" kann diese Aufgabe der Vervollständigung überlassen werden. Bei der Modellierung von Platzhalter wäre ein einfaches Austauschen nicht möglich, da ein typisierter Platzhalter nicht garantiert durch den neuen Cloud-Anbieter erfüllt werden kann.

4.3.5. Kombination der Anwendungsfälle

Die in den vorigen Abschnitten getrennt beschriebenen Anwendungsfälle können beliebig kombiniert werden. Eine mögliche Kombination wird in folgender Abbildung dargestellt. Dabei wird durch ein Requirement festgelegt, dass eine MySQL-Datenbank mit einem MySQL-Datenbank-Management-System verbunden werden muss. Der verwendete Webserver, der als Container für die PHP-Anwendung dient, wird durch einen Platzhalter abstrahiert. Ob dieser direkt mit der Cloud-Plattform verbunden wird oder ob weitere Komponenten dazwischen liegen wird durch die Verwendung eines Relationship Templates des Typs „Deferred“ offen gelassen.

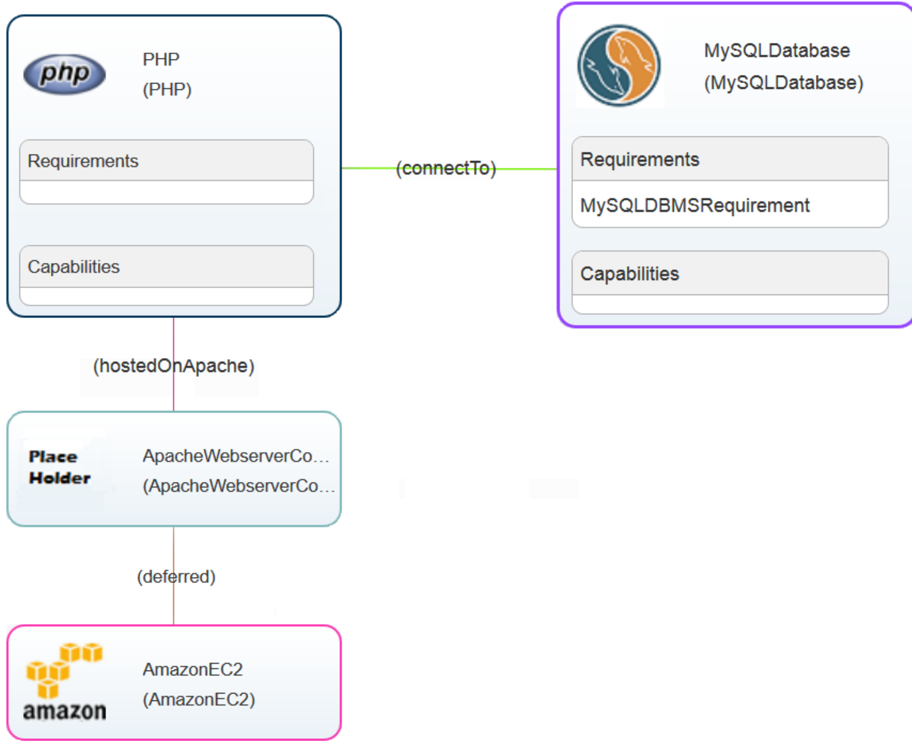

Abbildung 4.6.: Mögliche Kombination der Anwendungsfälle

4.3.6. Zusammenfassung

Unvollständige TOSCA-Topologien können den beschriebenen Anwendungsfällen entsprechend modelliert werden. Folgende Definitionen wurden im Rahmen des Konzeptes eingeführt und werden für die weiteren Kapitel als Grundlage verwendet:

Definition I: Vollständige TOSCA-Topologie:
Eine TOSCA-Topologie ist genau dann vollständig, wenn alle Requirements der Node Templates und der verwendeten Node Types erfüllt sind sowie keine Platzhalter-Node Templates und Relationship Templates des Typs „Deferred“ enthalten sind.

Definition II: Provisionierbare TOSCA-Topologie:
Eine Topologie ist genau dann provisionierbar, wenn diese weder Platzhalter-Node Templates noch Relationship Templates des Typs „Deferred“ enthält sowie alle Komponenten und Relationen, die für die Provisionierung einer Anwendung notwendig sind. Die Provisionierbarkeit einer Topologie ist dabei abhängig von der jeweiligen TOSCA-Laufzeitumgebung.

Dabei haben die Definitionen keinen Einfluss aufeinander. Eine Topologie kann unvollständig und dennoch provisionierbar sein, aber auch vollständig und nicht provisionierbar. Zum Beispiel können bestimmte Laufzeitumgebungen auch mit unerfüllten Requirements umgehen, sodass eine aufgrund von Requirements unvollständige Topologie trotzdem als provisionierbar definiert werden kann.

Die Provisionierbarkeit einer Topologie hängt wie bereits diskutiert von der jeweiligen TOSCA-Laufzeitumgebung ab. Die im nächsten Kapitel vorgestellten Konzepte sind jedoch dafür ausgelegt, auch für später standardisierte Typen anwendbar zu sein. In diesem Fall definiert sich die Provisionierbarkeit gemäß der Regeln sowie durch Node und Relationship Typen, die durch eine etwaige Standardisierung festgelegt werden.

4.4. Vervollständigung von TOSCA-Topologien

Nachdem im vorigen Kapitel die Anwendungsfälle bei der Modellierung unvollständiger bzw. nicht provisionierbarer Topologien beschrieben wurden, wird in diesem Kapitel die Vervollständigung dieser zu provisionierbaren Topologien erläutert.

Unter einer „Vervollständigung“ von TOSCA-Topologien wird das Ergänzen von nicht provisionierbaren Topologien durch Komponenten und Relationen (Node und Relationship Templates) verstanden, bis diese auf einer TOSCA-Laufzeitumgebung provisioniert werden können. Als Grundlage für die Vervollständigung dient eine nach 4.3 modellierte, nicht provisionierbare TOSCA-Topologie, verpackt als CSAR. Nach der Beschreibung von Anforderungen und Voraussetzungen an die Vervollständigung sowie eines Repository, welches der Speicherung verwendeter Typen wie Node Types, Relationship Types sowie Requirements und Capabilities dient, werden die Algorithmen der Vervollständigung vorgestellt. Diese vervollständigen eine Topologie in einem Schritt (siehe 4.1). Anschließend daran wird in 4.4.5 der Ablauf der Vervollständigung bei einer schrittweisen Vorgehensweise erläutert.

4.4.1. Anforderungen und Voraussetzungen

Um eine TOSCA-Topologie korrekt zu einer provisionierbaren Topologie vervollständigen zu können, müssen folgende Voraussetzungen erfüllt sein:

- Eine TOSCA-Topologie wurde wie in Kapitel 4.3 beschrieben modelliert.
- Die TOSCA-Topologie enthält valides, schemakonformes XML.
- Die bei der Modellierung verwendeten TOSCA-Requirements und Platzhalter stammen aus einer bekannten Menge.

4.4.2. Type-Repository

Bei der Vervollständigung von TOSCA-Topologien werden, wie bereits erläutert, Node und Relationship Templates zu einer Topologie hinzugefügt, bis diese auf einer TOSCA-Laufzeitumgebung provisioniert werden kann. Derartige Templates werden aus Node bzw. Relationship Types instanziiert. Daher muss eine Basis von Typen geschaffen werden, aus denen die einzufügenden Elemente instanziiert werden können. Um den Zugriff auf diese Typen zu vereinfachen, werden sie in einem Repository abgelegt, welches einfache Schnittstellen-Methoden für den Zugriff bietet. Dies fördert darüber hinaus eine einfache Erweiterbarkeit der Typ-Menge. Im Rahmen des Modellierungs-Tools Winery wurde ein derartiges Repository bereits implementiert. Der Zugriff auf das Winery-Repository geschieht über eine REST (Representational State Transfer)-Schnittstelle, die im Laufe der Vervollständigung genutzt wird. Neben den Node und Relationship Types sind im Winery Repository alle in TOSCA vorhandenen Typen abgespeichert (Policy Types, Artifact Types etc.). Dabei können die enthaltenen Typen per XML oder CSAR exportiert werden. Das vorgestellte Repository

dient als Grundlage für die im Folgenden beschriebenen Algorithmen der Vervollständigung.

Abbildung 4.7 zeigt die grafische Oberfläche des Winery Repositorys.

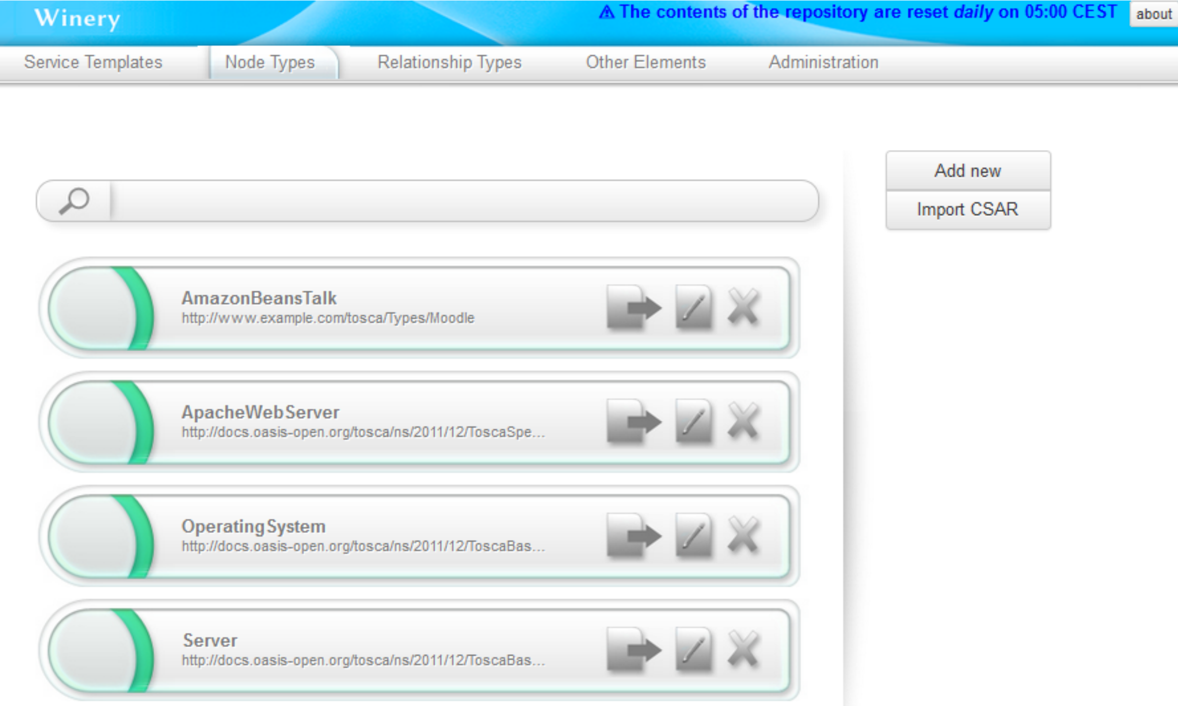

Abbildung 4.7.: Winery Repository

4.4.3. Algorithmen der Topologie-Vervollständigung

In diesem Abschnitt werden die Algorithmen der Vervollständigung dargestellt. Diese werden auf Basis der in Kapitel 4.3 dargestellten, nicht provisionierbaren TOSCA-Topologien beschrieben. Das Erfüllen der Eigenschaften (Property Definitions) einzelner Node Templates wird im Rahmen der Topologie-Vervollständigung nicht berücksichtigt (siehe Ausblick-Kapitel 7.2). Um die Komplexität zu verringern, gehen folgende Algorithmen von einer eindeutigen Lösung der Vervollständigung aus. Wie im Falle mehrerer möglicher Lösungen vorzugehen ist, wird anschließend in 4.4.4 erläutert. Des Weiteren wird pro Vervollständigung von genau einer verwendeten TOSCA-Laufzeitumgebung ausgegangen.

Um die Nachvollziehbarkeit zu erhöhen wurde der Algorithmus der Vervollständigung in mehrere Unteralgorithmen aufgespalten. Diese werden vom Hauptalgorithmus „TopologyCompletion“ aufgerufen. Die definierten Unteralgorithmen sind „FulfillRequirements“, „ReplacePlaceholders“ und „ProcessDeferredRelationshipTemplates“, die dem Erfüllen von Requirements, Platzhaltern und Relationship Templates des Typs „Deferred“ dienen. Diese Unteralgorithmen werden, anschließend an den Hauptalgorithmus, in den Unterkapiteln 4.4.3.5, 4.4.3.6 und 4.4.3.7 genauer beschrieben.

Des Weiteren werden Hilfsalgorithmen benötigt, die vermehrt durch diese Algorithmen aufgerufen werden. Die Hilfsalgorithmen werden im nächsten Abschnitt erläutert. Anschließend werden notwendige Grundkonzepte für die Vervollständigung dargestellt, die als Basis des Hauptalgorithmus „TopologyCompletion“ dienen. Dieser wird daraufhin in 4.4.3.4 beschrieben.

4.4.3.1 Hilfsalgorithmen

Bevor der Hauptalgorithmus „TopologyCompletion“ beschrieben werden kann, müssen zwei Hilfsalgorithmen eingeführt werden, die von diesem vermehrt genutzt werden. Dabei handelt es sich um den Algorithmus „SearchRelationshipType“, der Relationship Types für eine Verbindung zu einem eingefügten Node Template ermittelt sowie um den Algorithmus „InstantiateTemplate“, der Node bzw. Relationship Templates aus gegebenen Typen instanziiert. Diese werden im Folgenden erläutert.

Hilfsalgorithmus „SearchRelationshipType“
Wie bereits beschrieben werden während der Vervollständigung Node Templates zur Topologie hinzugefügt. Nach dem Einfügen eines Node Templates wird dieses mit der bestehenden Topologie mittels Relationship Templates verbunden. Jedoch stellt sich die Suche nach einer Verbindung als schwierig dar, da hierfür ein Relationship Type gefunden werden muss, aus dem ein passendes Relationship Template instanziiert werden kann. Dabei kommen oftmals mehrere Relationship Types für eine Verbindung infrage.
Um diese Probleme zu lösen wurde folgender Algorithmus für die Suche eines oder mehrerer passender Relationship Types entwickelt.

Für die Verbindung zweier Node Templates wird ein Relationship Type unter Angabe der zu verbindenden Node Templates wie folgt gesucht:

Algorithmus 4.1 Pseudo-Code: Suche passende Verbindungen zweier Node Templates

```
1: procedure SearchRelationshipType(NodeTemplate source, NodeTemplate target)
2:     List<RelationshipType> suitableRelationshipTypes = new List<>();
3:     for all relationshipTypes in repository do
4:         if (relationshipType.ValidSource == source.Type AND
5:             relationshipType.ValidTarget == target.Type) OR
6:             (relationshipType.ValidSource == source.Requirement.Type AND
7:             relationshipType.ValidTarget == target.Capability.Type) then
8:             suitableRelationshipTypes.add(relationshipType);
9:         end if
10:    end for
11: end procedure
```

Dabei werden alle Relationship Types im Repository durchlaufen und es wird überprüft, ob deren „Valid Source“ bzw. „Valid Target“-Elemente den Typen der zu verbindenden Node Templates oder der enthaltenen Requirements und Capabilities entspricht. Falls dies der Fall ist, wird der Relationship Type zur Menge möglicher Verbindungen aufgenommen. Findet sich kein solcher Relationship Type, werden generische Relationship Types ohne die optionalen „Valid Source“ und „Valid Target“-Elemente eingefügt.

Hilfsalgorithmus „InstantiateTemplate“

Dieser Algorithmus dient der Instanziierung von Node und Relationship Templates. Hierfür wird diesem ein Element des Typs EntityType übergeben, von dem sowohl Node Types als auch Relationship Types ableiten. Anschließend wird je nach Typ ein neues Node oder Relationship Template angelegt und dieses mit einer ID, einem Namen und einer Referenz auf den Typ versehen. Das instanziierte Template wird anschließend ausgegeben. Der Algorithmus wird nachfolgend in Pseudo-Code beschrieben:

Algorithmus 4.2 Pseudo-Code: Instanziierung von Templates

```
1:  procedure InstantiateTemplate(EntityType entity)
2:      if entity instanceof NodeType then
3:          NodeTemplate instantiatedNodeTemplate = new NodeTemplate();
4:          instantiatedNodeTemplate.setId(randomId);
5:          instantiatedNodeTemplate.setName(entity.getName());
6:          instantiatedNodeTemplate.setType(entity);
7:          return instantiatedNodeTemplate;
8:      else if entity instanceof RelationshipType then
9:          RelationshipTemplate instantiatedRelationshipTemplate =
10:             new RelationshipTemplate();
11:         instantiatedRelationshipTemplate.setId(randomId);
12:         instantiatedRelationshipTemplate.setName(entity.getName());
13:         instantiatedRelationshipTemplate.setType(entity);
14:         return instantiatedRelationshipTemplate;
15:     end if
16: end procedure
```

Neben den Hilfsalgorithmen ist für die Durchführung des „TopologyCompletion“-Algorithmus die Einführung zweier Modi notwendig:

4.4.3.2 Vervollständigungs-Modi

Bei der Vervollständigung von TOSCA-Topologien werden Requirements einer Topologie erfüllt. Unter Umständen ist dies jedoch nicht notwendig, wenn die verwendete TOSCA-Laufzeitumgebung diese bereits verarbeiten kann. Aus diesem Grund werden zwei verschiedenen Modi eingeführt, die in diesem Abschnitt beschrieben werden. Der gewählte Modus wird an den Hauptalgorithmus der Vervollständigung übergeben.

- FulfillAllRequirements-Modus
  In diesem Modus werden alle Requirements einer Topologie durch die Vervollständigung erfüllt, unabhängig davon, ob diese durch die TOSCA-Laufzeitumgebung verarbeitet werden können.
- FulfillEssentialRequirements-Modus
  Wird die Vervollständigung in diesem Modus durchgeführt, werden nur die Requirements erfüllt, die nicht durch die TOSCA-Laufzeitumgebung verarbeitet werden können.

4.4.3.3 Grundkonzept: Untersuchung auf Provisionierbarkeit

Bei den dargestellten Algorithmen der Vervollständigung werden Node Templates oder ganze Topologien nach Definition II auf Provisionierbarkeit überprüft. Der Ablauf dieser Überprüfung wird in diesem Abschnitt beschrieben. Zur Überprüfung wird eine vom Institut für Architektur von Anwendungssystemen entwickelte API genutzt, die als Provisioning-API bezeichnet wird. Diese API wird neben den im Folgenden beschriebenen Funktionen hauptsächlich dazu verwendet, ein CSAR direkt aus Winery in einer TOSCA-Laufzeitumgebung zu provisionieren. Dabei existiert eine Provisioning-API für alle angebundenen Laufzeitumgebungen. Die Provisioning-API bietet folgende Methoden an, die während der im Anschluss beschriebenen Vervollständigung aufgerufen werden:

- isTopologyProvisionable | Eingabe TOSCA-Topologie:
  Prüft ob eine TOSCA-Topologie auf einer TOSCA-Laufzeitumgebung provisioniert werden kann.
- isNodeTemplateProvisionable | Eingabe TOSCA Node Template; TOSCA Topologie:
  Prüft ob ein einzelnes Node Template einer Topologie auf einer TOSCA-Laufzeitumgebung provisioniert werden kann.
- findPossibleContainers | Eingabe TOSCA Node Template; TOSCA Topologie:
  Liefert ein oder mehrere Topologien, auf denen ein Node Template provisioniert werden kann. Liefert diese Methode eine leere Liste zurück, ist das verwendete Node Template nicht provisionierbar. Als Container wird in diesem Zusammenhang ein Topology Template bezeichnet, auf dem ein Node Template provisioniert werden kann.
- canHandleRequirement | Eingabe TOSCA Requirement:
  Prüft ob ein Requirement durch die TOSCA-Laufzeitumgebung verarbeitet werden kann.

Somit können mit Hilfe der Provisioning-API sowohl einzelne Node Templates und Requirements, als auch gesamte Topologien auf Provisionierbarkeit überprüft sowie passende Container für eine Provisionierung abgefragt werden.

4.4.3.4 „TopologyCompletion" - Hauptalgorithmus der Vervollständigung

Nachdem in den vorigen Abschnitten notwendige Voraussetzungen, Grundkonzepte und Hilfsalgorithmen definiert wurden, wird in diesem Kapitel der Hauptalgorithmus für die Vervollständigung einer TOSCA-Topologie dargestellt. Bei den in 4.3 behandelten Anwendungsfällen der Modellierung unvollständiger Topologien wurden Requirements, Platzhalter und Relationship Templates des Typs „Deferred" in eine Topologie eingefügt, um auf eine möglicherweise notwendige Vervollständigung hinzuweisen. Diese Elemente müssen durch den Algorithmus der Vervollständigung erfüllt werden. Hierfür werden die Unteralgorithmen „FulfillRequirements", „ReplacePlaceholders" und „ProcessDeferredRelationshipTemplates" verwendet.

Die dargestellten Algorithmen werden von der verwendeten TOSCA-Laufzeitumgebung beeinflusst, da die Provisionierbarkeit einer Topologie von dieser abhängt. Daher wird im Folgenden davon ausgegangen, dass der Anwender vor der Vervollständigung eine TOSCA-Laufzeitumgebung auswählt. Aus dieser wird anschließend ein Laufzeitumgebung-spezifisches-Objekt der Provisioning-API (siehe 4.4.3.3) instanziiert, welches an die Algorithmen übergeben wird. Des Weiteren wird bei folgenden Algorithmen davon ausgegangen, dass jeweils genau ein Node- oder Relationship Template eingefügt werden kann, um ein Requirement, einen Platzhalter oder ein Relationship Template des Typs „Deferred" zu erfüllen. Wie bei mehreren möglichen Templates vorgegangen wird, wird anschließend in 4.4.4 beschrieben. Das Ergebnis des Algorithmus ist eine provisionierbare Topologie. Ob die ausgegebene Topologie neben der Provisionierbarkeit auch vollständig sein muss, hängt von der verwendeten TOSCA-Laufzeitumgebung ab.

Der Hauptalgorithmus „TopologyCompletion" gliedert sich in 3 Phasen. Als erstes wird die Modellierungsphase durchgeführt, die in Kapitel 4.3 bereits erläutert wurde. In dieser modelliert der Modellierer eine nicht provisionierbare Topologie, die anschließend durch den Algorithmus vervollständigt werden soll. In der anschließenden Vervollständigungsphase wird diese Topologie dann durch TOSCA-Elemente (Node und Relationship Templates) - abhängig von der TOSCA-Laufzeitumgebung - zu einer vollständigen Topologie ergänzt. Nachdem die Vervollständigungsphase abgeschlossen ist, ist die Topologie jedoch noch nicht unbedingt provisionierbar. Daher wird anschließend die Provisionierbarkeit der Topologie über das erzeugte Objekt der Provisioning-API überprüft und gegebenenfalls weitere Komponenten hinzugefügt (siehe 4.4.3.3).

Der genaue Ablauf des Algorithmus und der enthaltenen Phasen wird im Folgenden schrittweise von der Modellierung bis zum vollständigen Modell beschrieben. Nach der textuellen Beschreibung wird der Algorithmus anschließend als Pseudo-Code dargestellt.

Als Eingabe für den Algorithmus dient eine nach 4.3 modellierte, nicht provisionierbare Topologie, ein Objekt der Provisioning-API (abhängig von der verwendeten TOSCA-Laufzeitumgebung) sowie der gewählte Modus (siehe 4.4.3.2).

Ablauf des Algorithmus „TopologyCompletion“:

- **Schritt 1**: Modellierungsphase: Der Modellierer modelliert eine TOSCA-Topologie nach 4.3 mittels XML-Editor oder einem grafischen TOSCA-Modellierungs-Tool.
- **Schritt 2:** Er verpackt die XML-Dateien anschließend manuell als CSAR oder nutzt den Export des Modellierungstools.
- **Schritt 3:** Die Vervollständigungs-Komponente importiert das CSAR.
- **Schritt 4:** Vervollständigungsphase:
  - **Schritt 4.1:** Durchführung des Algorithmus „ReplacePlaceholders“ (4.4.3.6).
  - **Schritt 4.2:** Durchführung des Algorithmus „ProcessDeferredRelationshipTemplates“ (4.4.3.7).
  - **Schritt 4.3:** Durchführung des Algorithmus „FulfillRequirements“ (4.4.3.5). Als Eingabe wird neben der nicht provisionierbaren Topologie der verwendete Modus (siehe 4.4.3.2) übergeben. Dieser beschreibt, ob alle Requirements der Topologie erfüllt werden sollen oder nur diejenigen, die nicht von der TOSCA-Laufzeitumgebung verarbeitet werden können.

    Anmerkung: Die Reihenfolge, in der die Algorithmen durchgeführt werden, ist für eine korrekte Vervollständigung wichtig. Bevor die Requirements einer Topologie erfüllt werden, müssen die Platzhalter und Deferred Relationship Templates durch das Einfügen von Node und Relationship Templates ersetzt werden. Hierbei können wiederum neue Requirements über deren Typen hinzu kommen.
- **Schritt 5:** Provisionierbarkeits-Überprüfung:
  - **Schritt 5.1:** Die Vervollständigungs-Komponente überprüft die vervollständigte Topologie auf Provisionierbarkeit in der verwendeten TOSCA-Laufzeitumgebung über die Provisioning-API.
    - **Schritt 5.1.1:** Falls provisionierbar, gehe zu Schritt 6.
    - **Schritt 5.1.1:** Falls nicht provisionierbar, gehe zu Schritt 5.2
  - **Schritt 5.2:** Für alle Node Templates wird an der Provisioning-API abgefragt, auf welchem Container es provisioniert werden kann. Diese gibt daraufhin ein Topology Template zurück.

    Anmerkung: Da das abgefragte Container-Template weitere Abhängigkeiten bzw. benötigte Komponenten besitzen kann, liefert die Provisioning-API ein Topology Template, welches mehrere Node Templates und deren Verbindungen enthalten kann. Somit besteht die Möglichkeit beliebig viele Node und Relationship Templates über die Provisioning-API zurückzugeben.
    - **Schritt 5.2.1:** Alle Node und Relationship Templates des erhaltenen Topology Templates werden in die bestehende Topologie eingefügt.

  - **Schritt 5.3:** Gehe zu Schritt 4.
    Dies ist notwendig, da durch das Hinzufügen von Node Templates Requirements der eingefügten Node Types hinzu kommen können, die noch nicht erfüllt wurden.

- **Schritt 6:** Die Vervollständigung wird abgeschlossen, die vervollständigte Topologie als CSAR verpackt und ausgegeben bzw. in das TOSCA-Modellierungstool importiert.

Ausgabe: Eine durch die verwendete Laufzeitumgebung provisionierbare TOSCA-Topologie.

Nachfolgend ist das Vorgehen als Algorithmus in Pseudo-Code beschrieben. Dabei wurde die Modellierungsphase bereits abgeschlossen und eine unvollständige, nicht provisionierbare Topologie an den Algorithmus übergeben:

Algorithmus 4.3 Pseudo-Code: Algorithmus „TopologyCompletion“

```
1:  // Instanziiert ein Objekt der API, abhängig von der verwendeten TOSCA-Runtime
2:  ProvisioningAPI provisioningAPI = new ProvisioningAPI(TOSCARuntime);
3:  procedure TopologyCompletion(Topology topology, ProvisioningAPI provisioningAPI,
    Mode completionMode)
4:      // Vervollständigungsphase
5:      ReplacePlaceholders(topology);
6:      ProcessDeferredRelationshipTemplates(topology, provisioningAPI);
7:      FulfillRequirements(topology, provisioningAPI, completionMode);
8:      // Prüfen der Provisionierbarkeit
9:      Boolean provisionable = provisioningAPI.isTopologyProvisionable(topology);
10:     if !provisionable then
11:         for all nodeTemplates in topology do
12:             Boolean nodeTemplateProvisionable =
13:                 provisioningAPI.isNodeTemplateProvisionable(nodeTemplate);
14:             if !nodeTemplateProvisionable then
15:                 TopologyTemplate container =
16:                     provisioningAPI.findPossibleContainers(nodeTemplate, topology);
17:                 for all elements in container do
18:                     topology.add(element);
19:                 end for
20:             end if
21:         end for
22:     else
23:         return topology;
24:     end if
25:     // Rekursiver Aufruf bis die Topologie auf der verwendeten
26:     // TOSCA-Laufzeitumgebung provisionierbar ist.
27:     TopologyCompletion(topology, provisioningAPI, completionMode)
28: end procedure
```

4.4.3.5 Algorithmus „FulfillRequirements“

Während der Vervollständigungsphase des Hauptalgorithmus wird der Unteralgorithmus „FulfillRequirements“ aufgerufen, um eventuelle Requirements der unvollständigen Topologie zu erfüllen. Diese wurden während der Modellierungsphase durch den Modellierer festgelegt, um Bedingungen an die zu ergänzende Infrastruktur zu stellen. Wie in 4.4.3.2 beschrieben, existieren zwei Modi, in denen die Vervollständigung durchgeführt werden kann. Diese legen fest, ob alle Requirements erfüllt werden sollen oder nur diejenigen, die nicht durch die verwendete TOSCA-Laufzeitumgebung verarbeitet werden können.

Der Algorithmus „FulfillRequirements“ spaltet sich in die Unteralgorithmen „AnalyzeRequirements“ und „MatchRequirementAndCapability“ auf, um die Komplexität zu verringern. Diese werden nachfolgend beschrieben:

Unteralgorithmus „AnalyzeRequirements“
Der folgende Algorithmus dient der Analyse von unerfüllten Requirements an Node Templates und Types. Im Rahmen der Requirement-Analyse werden alle vorhandenen Requirements einer Topologie mit den dazugehörigen Node Templates in einer Key-Value-Map abgespeichert, wobei der Schlüssel ein Requirement, der Wert ein Node Template darstellt. Dies macht es während der Vervollständigung möglich, jedes Requirement einem bestimmten Node Template zuzuordnen. Dabei werden sowohl die Requirement Definitions der Node Types, als auch die Requirements der Node Templates zur beschriebenen Map hinzugefügt. Die Analyse wird in Algorithmus 4.4 in Pseudo-Code dargestellt.

Algorithmus 4.4 Pseudo-Code: Analyse von Requirements

```
1: procedure AnalyzeRequirements(Topology topology)
2:     Map<Requirement, NodeTemplate> foundRequirements = new Map<>();
3:     for all nodeTemplates in topology do
4:         if nodeTemplate.getRequirements() != null then
5:             // Füge Requirement Definitions des Node Types zur Map hinzu
6:             for all requirements in nodeTemplate.NodeType do
7:                 if !requirement.fulfilled then
8:                     foundRequirements.add(requirement, nodeTemplate);
9:                 end if
10:            end for
11:            // Füge Requirements des Node Templates zur Map hinzu
12:            for all requirements in NodeTemplate do
13:                if !requirement.fulfilled then
14:                    foundRequirements.add(requirement, nodeTemplate);
15:                end if
16:            end for
17:        end if
18:    end for
19: end procedure
```

Unteralgorithmus „MatchRequirementAndCapability“
Neben dem Analyse-Algorithmus, der Node Templates und Types auf unerfüllte Requirements analysiert, wird der Algorithmus „MatchRequirementAndCapability“ benötigt, um einen Node Type zu finden, dessen Capability ein gegebenes Requirement erfüllen kann. Hierfür wird zuerst der Typ des Requirements untersucht. Dieser enthält das Attribut „requiredCapabilityType“, welches angibt, welcher Capability Type dieses Requirement erfüllen kann. Anschließend werden alle Node Types durchlaufen und es wird überprüft, ob diese den geforderten Capability Type besitzen. Ist dies der Fall, kann der Node Type das Requirement erfüllen und wird zu einer Liste passender Node Types aufgenommen.
Der Algorithmus wird im Folgenden in Pseudo-Code dargestellt.

Algorithmus 4.5 Pseudo-Code: Matching von Requirements und Capabilities

```
 1: procedure MatchRequirementAndCapability(Requirement requirement)
 2:     List<NodeType> matchingNodeTypes = new List<>();
 3:
 4:     CapabilityType requiredCapabilityType = requirement.Type.RequiredCapabilityType;
 5:     for all nodeTypes in repository do
 6:         for all capabilityTypes in nodeType do
 7:             if capabilityType == requiredCapabilityType then
 8:                 matchingNodeTypes.add(nodeType);
 9:             end if
10:         end for
11:     end for
12: end procedure
```

Ablauf des Algorithmus „FulfillRequirements“
Dieser Algorithmus dient dem Erfüllen von Requirements einer Topologie. Dabei werden die eben beschriebenen Unteralgorithmen aufgerufen.

Eingabe: Eine auf der verwendeten Laufzeitumgebung nicht provisionierbare TOSCA-Topologie, ein Objekt der Provisioning-API (abhängig von der TOSCA-Laufzeitumgebung) sowie der Modus, mit dem die Topologie erfüllt werden soll.

Vorgehen:

- **Schritt 1:** Untersuchung des Modus:
  - **Schritt 1.1:** Falls die Vervollständigung im Modus „FulfillEssentialRequirements“ durchgeführt wird, d.h. nur die Requirements erfüllt werden sollen, die nicht durch die verwendete TOSCA-Laufzeitumgebung verarbeitet werden können, gehe zu Schritt 2.
  - **Schritt 1.2:** Falls die Vervollständigung im Modus „FulfillAllRequirements“ durchgeführt wird, d.h. alle vorhandenen Requirements durch die Vervollständigung erfüllt werden sollen, gehe zu Schritt 3.

- **Schritt 2:** Für alle Node Templates wird an der Provisioning-API abgefragt, ob dieses bereits in der verwendeten Laufzeitumgebung provisioniert werden kann.
  - **Schritt 2.1:** Falls provisionierbar werden alle Requirements des Node Templates und Types - falls vorhanden - als erfüllt vermerkt.
  - **Schritt 2.1:** Falls nicht provisionierbar wird für jedes Requirement des Templates bzw. Types abgefragt, ob die TOSCA-Laufzeitumgebung dieses verarbeiten kann. Falls ja, wird das Requirement als erfüllt vermerkt. Dadurch werden anschließend nur die Requirements erfüllt, die nicht durch die TOSCA-Laufzeitumgebung verarbeitet werden können.
- **Schritt 3:** Über den Algorithmus „AnalyzeRequirements" werden sowohl die Templates der Topologie als auch deren Typen auf die Existenz von unerfüllten Requirements analysiert. Die gefundenen Requirements werden mit einer Referenz auf das zugehörige Node Template abgespeichert.
  - **Schritt 3.1:** Falls keine unerfüllten Requirements in der Topologie existieren, gehe zu Schritt 6.
  - **Schritt 3.2:** Falls unerfüllte Requirements existieren, gehe zu Schritt 4.
- **Schritt 4:** Für jedes gefundene, nicht als erfüllt vermerkte, Requirement:
  - **Schritt 4.1:** Mittels Algorithmus „MatchRequirementAndCapability" wird nach einem passenden Node Type gesucht, um das Requirement zu erfüllen.
  - **Schritt 4.2:** Aus dem erhaltenen Typ wird durch Algorithmus „InstantiateTemplate" ein Node Template instanziiert.
  - **Schritt 4.3:** Mittels Algorithmus „SearchRelationshipType" wird ein Relationship Type gesucht, um zu dem eingefügten Node Template zu verbinden. Aus diesem wird daraufhin durch den „InstantiateTemplate"-Algorithmus ein Relationship Template instanziiert.
  - **Schritt 4.4:** Die instanziierten Templates werden in die Topologie eingefügt und das Requirement anschließend als erfüllt vermerkt.
- **Schritt 5:** Gehe zu Schritt 1, um eventuell neu hinzugekommene Requirements zu erfüllen.
- **Schritt 6:** Die Topologie wird an den Algorithmus der Vervollständigung zurückgegeben.

Ausgabe: Eine Topologie, die keine unerfüllten Requirements mehr enthält.

Im Folgenden ist das Vorgehen als Algorithmus in Pseudo-Code beschrieben:

Algorithmus 4.6 Pseudo-Code: Algorithmus „FulfillRequirements“

```
1:  procedure FULFILLREQUIREMENTS(Topology topology, ProvisioningAPI provisioningAPI,
    Mode completeAllRequirements)
2:      // Überprüfe den Modus, markiere gegebenenfalls Requirements als erfüllt,
3:      // die bereits von der TOSCA-Laufzeitumgebung verarbeitet werden können
4:      if mode == FulfillEssentialRequirements then
5:          for all nodeTemplates in topology do
6:              if provisioningAPI.isNodeTemplateDeployable(nodeTemplate) then
7:                  for all requirements in nodeTemplate do
8:                      requirement.fulfilled = true;
9:                  end for
10:             else
11:                 for all requirements in nodeTemplate do
12:                     if provisioningAPI.canHandleRequirement(requirement) then
13:                         requirement.fulfilled = true;
14:                     end if
15:                 end for
16:             end if
17:         end for
18:     end if
19:     // Suche an Node Templates und Node Types vorhandene Requirements und
20:     // speichere diese in einer Map mit Referenz auf das Template.
21:     Map<Requirement, NodeTemplate> requirements = AnalyzeRequirements(topology);
22:     if requirements.Empty then
23:         return topology;
24:     else
25:         for all requirements do
26:             if !requirement.fulfilled then
27:                 // Füge ein NodeTemplate ein, welches das Requirement erfüllt
28:                 NodeType suitableNodeType = MatchRequirementAndCapability(requirement);
29:                 NodeTemplate instance = InstantiateTemplate(suitableNodeType);
30:                 topology.add(instance);
31:
32:                 // Füge ein Verbindung zum eingefügten NodeTemplate ein
33:                 RelationshipType relationshipType =
34:                     SearchRelationshipType(nodeTemplate, instance);
35:                 topology.add(InstantiateTemplate(relationshipType));
36:
37:                 requirement.fulfilled = true;
38:             end if
39:         end for
40:     end if
41:     // Rekursiver Aufruf, da durch das Einfügen von Node Templates weitere
42:     // Requirements hinzukommen können.
43:     FulfillRequirements(topology, completeAllRequirements);
44: end procedure
```

4.4.3.6 Algorithmus „ReplacePlaceholders“

Der Modellierer kann bei der Topologie-Modellierung einen oder mehrere Platzhalter verwendet haben, um Teile der Infrastruktur oder der Anwendung zu abstrahieren. Derartige Platzhalter sollen im Rahmen des Algorithmus „ReplacePlaceholders“ durch konkrete Node Templates ersetzt werden. Hierbei werden als erstes zwei Unteralgorithmen definiert. Der Algorithmus „AnalyzePlaceholders“ dient dem Auffinden von Platzhaltern einer Topologie. Der Unteralgorithmus „SearchPlaceholderReplacement“ wird für die Suche nach einem Node Template, das einen Platzhalter ersetzen kann, verwendet.

Unteralgorithmus „AnalyzePlaceholders“
Um die Platzhalter in einer Topologie aufzufinden wird der Algorithmus „AnalyzePlaceholders“ eingeführt. Dieser erhält die Topologie als Eingabe, durchläuft alle vorhandenen Node Templates und überprüft, ob deren Typ vom abstrakten Typ „Placeholder“ abgeleitet ist. Falls dies der Fall ist wird das Node Template zur Liste gefundener Platzhalter hinzugefügt.

Nachfolgend ist das Vorgehen als Algorithmus in Pseudo-Code beschrieben:

Algorithmus 4.7 Pseudo-Code: Analyse der Platzhalter

```
 1: procedure AnalyzePlaceholders(Topology topology)
 2:     List<NodeTemplate> foundPlaceholders = new List<>();
 3:
 4:     for all nodeTemplates in topology do
 5:         // Placeholder steht hier für den abstrakten Platzhalter-Typ
 6:         if nodeTemplate.Type.DerivedFrom == Placeholder then
 7:             foundPlaceholders.add(nodeTemplate);
 8:         end if
 9:     end for
10: end procedure
```

Unteralgorithmus „SearchPlaceholderReplacement“
Während der Vervollständigung von Topologien werden Platzhalter durch passende Node Types ersetzt. Das Auffinden dieser Node Types wird in diesem Abschnitt geschildert. Nachdem ein Node Template über dessen abstrakten Typ als Platzhalter identifiziert wurde, wird nach Node Types im Repository gesucht, die diesen ersetzen können. Wie bereits in 4.3.3 beschrieben, sind die Platzhalter typisiert. Beispielsweise können Platzhalter des Typs „Betriebssystem“, „Webserver“ o.ä. existieren. Die Bezeichnungen der Platzhalter-Typen stellen dabei Obertypen konkreter Node Types dar, die für eine Ersetzung des Platzhalters verwendet werden können. Existiert beispielsweise ein Platzhalter des Typs „Betriebssystem“, kann dieser durch alle Node Types mit dem Obertyp „Betriebssystem“ ersetzt werden.

Nachfolgend ist das Vorgehen als Algorithmus in Pseudo-Code beschrieben:

Algorithmus 4.8 Pseudo-Code: Algorithmus SearchPlaceholderReplacement

```
1: procedure SearchPlaceholderReplacement(NodeTemplate placeholder)
2:     List<NodeType> matchingNodeTypes = new List<>();
3:
4:     for all nodeTypes in repository do
5:         if nodeType.DerivedFrom == placeholder.Type then
6:             matchingNodeTypes.add(nodeType);
7:         end if
8:     end for
9: end procedure
```

Ablauf des Algorithmus „ReplacePlaceholders“:

Eingabe: Eine auf der verwendeten Laufzeitumgebung nicht provisionierbare TOSCA-Topologie.

Vorgehen:

- **Schritt 1:** Die Platzhalter der Topologie werden mittels Algorithmus „AnalyzePlaceholders“ analysiert.
  - **Schritt 1.1:** Falls keine Platzhalter existieren, gehe zu Schritt 3.
  - **Schritt 1.2:** Falls Platzhalter existieren, gehe zu Schritt 2.
- **Schritt 2:** Für alle Platzhalter wird mittels Algorithmus „SearchPlaceholderReplacement“ nach einem passenden Node Type für eine Ersetzung gesucht.
  - **Schritt 2.1:** Aus diesem wird mittels „InstantiateTemplate“ ein Node Template instanziiert und der Platzhalter durch dieses ersetzt.
  - **Schritt 2.2:** Durch Algorithmus „SearchRelationshipType“ wird nach einem Relationship Type gesucht, um zu dem eingefügten Node Template zu verbinden und aus diesem mittels „InstantiateTemplate“ ein Relationship Template instanziiert. Das generische Relationship Template „depends on“, das für die Verbindung zum Platzhalter verwendet wurde, wird anschließend aus der Topologie entfernt. Analog hierzu werden anschließend Relationship Templates für die ausgehenden Verbindungen des Platzhalters instanziiert. Die generischen, ausgehenden „depends on“-Verbindungen des Platzhalters werden anschließend ebenfalls aus der Topologie entfernt.
  - **Schritt 2.3:** Die instanziierten Templates werden in die Topologie eingefügt.
- **Schritt 3:** Die Topologie wird an den Algorithmus der Vervollständigung zurückgegeben.

Ausgabe: Eine Topologie, die keine Platzhalter mehr enthält.

Nachfolgend ist das Vorgehen als Algorithmus in Pseudo-Code beschrieben:

Algorithmus 4.9 Pseudo-Code: Algorithmus der Vervollständigung von Platzhaltern

```
 1: procedure REPLACEPLACEHOLDERS(Topology topology)
 2:     List<Requirement> placeholders = AnalyzePlaceholders(topology);
 3:
 4:     for all placeholders do
 5:         // Suche ein- und ausgehende Verbindungen des Platzhalters
 6:         List<RelationshipTemplate> placeholderConnections = new List<>();
 7:         for all relationshipTemplates in topology do
 8:             if relationshipTemplate.Source == placeholder OR
 9:                 relationshipTemplate.Target == placeholder then
10:                 placeholderConnections.add(relationshipTemplate)
11:             end if
12:         end for
13:
14:         // Suche zu einem Platzhalter passenden NodeType und instanziiere Template
15:         NodeType suitableNodeType = SearchPlaceholderReplacement(placeholder);
16:         NodeTemplate instance = InstantiateTemplate(suitableNodeType);
17:         topology.add(instance);
18:
19:         // Füge ein- und ausgehende Verbindungen des eingefügten Node Templates ein
20:         for all relationshipTemplates in placeholderConnections do
21:             if relationshipTemplate.Target == placeholder then
22:                 // Eine eingehende Verbindung des Platzhalters wird erstellt
23:                 RelationshipType relationshipType =
24:                     SearchRelationshipType(relationshipTemplate.Source, instance);
25:             else if relationshipTemplate.Source == placeholder then
26:                 // Eine ausgehende Verbindung des Platzhalters wird erstellt
27:                 RelationshipType relationshipType =
28:                     SearchRelationshipType(relationshipTemplate.Target, instance);
29:             end if
30:
31:             topology.add(InstantiateTemplate(relationshipType));
32:             // Entferne den Platzhalter und dessen Verbindungen
33:             topology.remove(relationshipTemplate);
34:             topology.remove(placeholder);
35:         end for
36:     end for
37:     return topology;
38: end procedure
```

4.4.3.7 Algorithmus „ProcessDeferredRelationshipTemplates“

Die Topologie kann ein oder mehrere Relationship Templates des Typs „Deferred“ enthalten. Die Vervollständigung dieser wird durch Algorithmus „ProcessDeferredRelationshipTemplates“ als Tiefensuche über alle möglichen einzufügenden Node Types durchgeführt, wobei ein Pfad zum Ziel-Template eines Relationship Templates des Typs „Deferred“ gesucht wird.

Anmerkung: Auch in diesem Algorithmus sind mehrere Lösungen d.h. Pfade zum Ziel-Template möglich. Dies wird wie in den vorigen Abschnitten ebenfalls nicht berücksichtigt, um die Komplexität zu verringern. Sind mehrere Topologie-Lösungen möglich, werden diese dem Modellierer nach der Vervollständigung zur Auswahl gegeben.

Wie in den vorigen Abschnitten wird für das Erfüllen von Relationship Templates des Typs „Deferred“ ebenfalls ein Unteralgorithmus „AnalyzeDeferredRelationshipTemplates“ eingeführt:

Unteralgorithmus „AnalyzeDeferredRelationshipTemplates“
Der Algorithmus „AnalyzeDeferredRelationshipTemplates“ dient der Analyse von Relationship Templates des Typs „Deferred“ einer Topologie. Dabei werden alle Relationship Templates durchlaufen und dessen Typ untersucht. Handelt es sich um den Typ „Deferred“, wird das Template zu einer Liste gefundener „Deferred“-Relationen aufgenommen.

Algorithmus 4.10 Pseudo-Code: Analyse von Relationship Templates des Typs „Deferred“

```
1: procedure ANALYZEDEFERREDRELATIONSHIPTEMPLATES(Topology topology)
2:     List<NodeTemplate> foundDeferredRelations = new List<>();
3:
4:     for all relationshipTemplates in topology do
5:         if relationshipTemplate.Type == „Deferred“ then
6:             foundDeferredRelations.add(relationshipTemplate);
7:         end if
8:     end for
9: end procedure
```

Bei folgendem Algorithmus wird zur Vereinfachung davon ausgegangen, dass das Ausgangs-Node Template eines Relationship Templates des Typs „Deferred“ maximal ein Requirement enthält.

Ablauf des Algorithmus „ProcessDeferredRelationshipTemplates“:

Eingabe: Eine auf der verwendeten Laufzeitumgebung nicht provisionierbare TOSCA-Topologie sowie ein Objekt der Provisioning-API (abhängig von der verwendeten TOSCA-Laufzeitumgebung).

Vorgehen:

- **Schritt 1:** Die Relationship Templates des Typs „Deferred“ werden mittels Algorithmus „AnalyzeDeferredRelationshipTemplates“ analysiert.
  - **Schritt 1.1:** Falls keine Relationship Templates des Typs „Deferred“ existieren, gehe zu Schritt 3.
  - **Schritt 1.2:** Falls Relationship Templates des Typs „Deferred“ existieren, gehe zu Schritt 2.
- **Schritt 2:** Für jedes „Deferred“-Relationship Template wird das Ausgangs-Node Template der Verbindung gesucht.
  - **Schritt 2.1:** Dieses Node Template wird auf Requirements untersucht.
    - **Schritt 2.1.1:** Falls ein Requirement existiert:
      - **Schritt 2.1.1.1:** Für dieses Requirement wird ein passender Node Type mittels Algorithmus „MatchRequirementAndCapability“ gesucht und aus diesem durch Algorithmus „InstantiateNodeTemplate“ ein Node Template instanziiert, welches in die Topologie eingefügt wird.
      - **Schritt 2.1.1.2:** Mittels Algorithmus „SearchRelationshipTemplate“ wird nach einem Relationship Type gesucht, um zu dem eingefügten Node Template zu verbinden. Aus diesem wird durch Algorithmus „InstantiateTemplate“ ein Relationship Template instanziiert und in die Topologie eingefügt.
    - **Schritt 2.1.2:** Falls kein Requirement existiert:
      - **Schritt 2.1.2.1** Für das Ausgangs-Node Template wird an der Provisioning-API abgefragt, auf welchem Container es provisioniert werden kann. Daraufhin gibt diese ein Topology Template zurück.
      - **Schritt 2.1.2.2:** Das erste Node Template des erhaltenen Topology Templates wird in die Topologie eingefügt. Mittels Algorithmus „SearchRelationshipTemplate“ wird anschließend nach einem Relationship Type gesucht, um zu dem eingefügten Node Template zu verbinden. Aus diesem wird

durch Algorithmus „InstantiateTemplate“ ein Relationship Template instanziiert und in die Topologie eingefügt.

- **Schritt 2.2:** Es wird überprüft, ob das eingefügte Node Template dem ursprünglichen Ziel des „Deferred“-Relationship Templates entspricht.
  - Falls nein, wird das eingefügte Node Template als Eingabe für 2.1. verwendet
  - Falls ja, wird dieses Node Template, dessen Verbindung sowie das „Deferred“-Relationship Template aus der Topologie gelöscht. Anschließend wird ein Relationship Type mittels Algorithmus „SearchRelationshipType“ gesucht, um zum ursprünglichen Ziel-Template zu verbinden. Aus diesem wird daraufhin ein Relationship Template instanziiert und in die Topologie eingefügt. Gehe anschließend zu Schritt 3.
- **Schritt 3:** Die Topologie wird an den Algorithmus der Vervollständigung zurückgegeben.

Ausgabe: Eine Topologie, die keine Relationship Templates des Typs „Deferred“ mehr enthält.

Nachfolgend ist das Vorgehen als Algorithmus in Pseudo-Code dargestellt. Dabei wird die Vervollständigung eines einzelnen Relationship Templates des Typs „Deferred“ beschrieben. Als Eingabe erhält der Algorithmus die Topologie, das Ausgangs-Node Template der Deferred-Verbindung (Source), das Ziel-Template zu dem verbunden wird (Target) sowie ein Objekt der Provisioning-API. Der Algorithmus sucht ausgehend vom Source-Template solange nach einzufügenden Node Templates, bis eine Verbindung zum Target-Template möglich ist. Dies wird als Tiefensuche über alle möglichen einzufügenden Node Templates realisiert.

Algorithmus 4.11 Pseudo-Code: Algorithmus der Vervollständigung von Relationship Templates des Typs „Deferred“

```
1:
2: NodeTemplate source = deferredRelationshipTemplate.Source;
3: NodeTemplate target = deferredRelationshipTemplate.Target;
4:
5: procedure PROCESSDEFERREDRELATIONSHIPTEMPLATES(TopologyTemplate topology, source, target, ProvisioningAPI provisioningAPI)
6:     // Speichere das eingefügte Node Template in dieser Variable
7:     NodeTemplate addedNodeTemplate;
8:     if source.hasRequirement() then
9:         // Füge ein NodeTemplate ein, welches das Requirement erfüllt
10:        NodeType suitableNodeType = MatchRequirementAndCapability(source.Requirement);
11:        addedNodeTemplate = InstantiateTemplate(suitableNodeType);
12:        topology.add(addedNodeTemplate);
13:    else
14:        TopologyTemplate container =
15:            provisioningAPI.findPossibleContainers(source, topology);
16:        addedNodeTemplate = container.FirstNodeTemplate;
17:        topology.add(addedNodeTemplate);
18:    end if
19:
20:    // Füge ein Verbindung zum eingefügten NodeTemplate ein
21:    RelationshipType relationshipType = SearchRelationshipType(source, addedNodeTemplate);
22:    RelationshipTemplate addedRelationshipTemplate =
23:        InstantiateTemplate(relationshipType);
24:    topology.add(addedRelationshipTemplate);
25:
26:    if addedNodeTemplate != target then
27:        // rekursiver Aufruf des Algorithmus
28:        processDeferredRelationshipTemplates(topology, addedNodeTemplate, target,
29:            provisioningAPI);
30:    else
31:        // Entferne das eingefügte Node Template und dessen Verbindung, da dieses
32:        // bereits durch das ursprüngliche Target-Template modelliert wurde
33:        topology.remove(addedNodeTemplate);
34:        topology.remove(addedRelationshipTemplate);
35:        // Entferne die Deferred-Verbindung
36:        topology.remove(deferredRelationshipTemplate);
37:
38:        // Verbinde zum ursprünglichen Ziel der Deferred-Verbindung
39:        RelationshipType type = SearchRelationshipType(source, target);
40:        topology.add(InstantiateNodeTemplate(type));
41:        return topology;
42:    end if
43: end procedure
```

4.4.4. Auswahl der Templates

In den beschriebenen Algorithmen wurde zur Vereinfachung davon ausgegangen, dass eine eindeutige Lösung für das Einfügen von Node oder Relationship Templates existiert. Dies ist jedoch in der Praxis nur selten der Fall. In diesem Abschnitt wird ein Konzept vorgestellt, wie einzufügende Node und Relationship Templates ausgewählt werden können.

4.4.4.1 Auswahl der Node Templates

Bei der Vervollständigung werden zu den modellierten Komponenten passende Node Templates eingefügt. Oftmals existieren jedoch mehrere Node Templates die dabei infrage kommen. Es ist außerdem möglich, dass statt einem Node Template ein vollständiges Topology Template eingefügt werden kann. Die Auswahl der einzufügenden Komponenten kann auf verschiedene Arten erfolgen:

- Nutzerabfrage
  Die passenden Node Templates werden dem Nutzer zur Auswahl gegeben. Die Entscheidung über die verwendete Komponente liegt bei ihm.
  - **Vorteil:** Reduzierter Aufwand bei der automatischen Vervollständigung.
  - **Nachteil:** Mehraufwand für den Nutzer.
- Programmatische Auswahl
  Als programmatische Auswahl wird eine automatisierte, rechnergesteuerte Selektion bezeichnet, die ohne Interaktion eines menschlichen Nutzers durchgeführt werden kann. Diese Auswahl kann auf zwei Arten geschehen:

  Priorisierte Auswahl: Die Node Templates werden mit Prioritäten versehen, die als Anhaltspunkt für die Auswahl der Komponenten verwendet werden.
  - **Vorteil:** Hohe Wahrscheinlichkeit für den Nutzer zufriedenstellende Topologien zu produzieren.
  - **Nachteil:** Einführen eines Prioritäten-Systems führt zu hohem Aufwand und verringert die Erweiterbarkeit der Typ-Menge.

  Zufällig: Die Auswahl erfolgt nach dem Zufallsprinzip.
  - **Vorteil:** Kaum Aufwand bei der Umsetzung.
  - **Nachteil:** Kann zu unerwünschten Ergebnissen führen.

Ergebnis: Existieren mehrere Möglichkeiten einzufügender Node Templates, wird im Rahmen dieser Arbeit eine Nutzerabfrage durchgeführt. Dabei wird bei der Vervollständigung in einem Schritt nicht für jedes Node Templates einzeln abgefragt, ob es eingefügt werden soll. Stattdessen werden Topologie-Kopien für jede mögliche Lösung an den Entscheidungspunkten erzeugt, aus denen der Nutzer nach der Vervollständigung auswählen kann.

4.4.4.2 Auswahl von Relationship Templates

Unter Umständen existieren mehrere Relationship Types, die für die Verbindung zweier Node Templates infrage kommen. Beispielsweise hat der Modellierer ein Node Template „WAR“ modelliert, welches er mit einer Webserver-Komponente „Apache Tomcat“ verbinden möchte. Für diese Verbindung können die Relationship Types „hosted on“, „deployed on“ und „runs in“ verwendet werden. Dabei können beliebig viele, mindestens jedoch eine der Verbindungen ausgewählt werden. Wichtig ist, dass die verwendete TOSCA-Laufzeitumgebung die Relationship Types verarbeiten kann.

Die Auswahl der Verbindung kann entweder automatisch oder nutzergesteuert erfolgen:

1. Nutzerabfrage
   Der Nutzer erhält eine Auswahl einzufügender Relationship Templates, aus denen er auswählen kann.
2. Automatische Auswahl
   - Füge alle möglichen Verbindungen in die Topologie ein.
   - Wähle zufällig eine der Verbindungen.
     - **Vorteil:** Ein automatischer Ansatz ist einfach umzusetzen.
     - **Nachteil:** Es ergibt sich ein Mehraufwand des Modellierers, da dieser unter Umständen weitere Relationship Templates hinzufügen oder löschen muss.

Ergebnis: Im Rahmen dieser Arbeit wird die Nutzerabfrage gewählt, da dies die einzige Vorgehensweise ist, die dem Modellierer ein zufriedenstellendes Ergebnis garantiert.

4.4.5. Schrittweise Vervollständigung

Wie bereits in 4.2 beschrieben soll neben der Vervollständigung in einem Schritt eine schrittweise Vervollständigung geschaffen werden, bei der dem Nutzer in jedem Schritt einzufügende Node und Relationship Templates zur Auswahl gegeben werden. Die Algorithmen aus den vorigen Abschnitten müssen hierfür lediglich leicht modifiziert werden. Kommen mehrere einzufügende Node oder Relationship Types infrage, werden aus diesen Templates instanziiert und dem Modellierer vor dem Einfügen in die Topologie zur Auswahl gegeben. Erst wenn sich der Modellierer für ein Template entschieden hat, wird dieses in die Topologie eingefügt. Das bedeutet, dass die Algorithmen der Vervollständigung an derartigen Entscheidungspunkten unterbrochen werden müssen. Nach der Auswahl werden die Algorithmen an der unterbrochenen Stelle fortgesetzt. Dies ermöglicht es eine nutzergesteuerte Topologie-Vervollständigung umzusetzen. Eine weitere Besonderheit der schrittweisen Vorgehensweise ist es, dem Modellierer die Möglichkeit zu geben, die Algorithmen zu unterbrechen und gegebenenfalls selbst Änderungen an der Topologie vorzunehmen. Durch diesen Ansatz kann - falls durch den Modellierer gewünscht - auf Platzhalter oder Relationship Templates des Typs „Deferred“ verzichtet werden.

4.5. Integration der Konzepte in Winery

Wie in den Einführungskapiteln beschrieben sollen die entstandenen Konzepte in das TOSCA-Modellierungstool Winery integriert werden, anstatt eine externe „stand-alone“-Lösung zu schaffen. Zwar bietet eine externe Lösung den Vorteil der Portierbarkeit und somit vielfältiger Einsatzmöglichkeiten, jedoch überwiegen die Vorteile der integrierten Lösung stark. Zum einen ist es durch eine integrierte Lösung nicht notwendig die Topologie zu exportieren und ein eigenes Datenmodell zu befüllen, zum anderen können die Methoden zur Anbindung an das Winery-Repository sowie zahlreiche in Winery implementierte Hilfsmethoden problemlos mitverwendet werden. Der hierbei gesparte Aufwand kann für eine umfangreichere Funktionalität genutzt werden. Darüber hinaus ist Winery im Moment eines der wenigen TOSCA-Modellierungstools und wird auch in Zukunft weiterentwickelt, sodass die Vervollständigung für die Nutzer von Winery eine große Bereicherung darstellen kann. Implementiert werden dabei sowohl die Vervollständigung in einem Schritt als auch die schrittweise Vervollständigung sowie eine Überprüfung auf Provisionierbarkeit einer Topologie. Die Integration der Vervollständigung in Winery ist in folgender Grafik dargestellt.

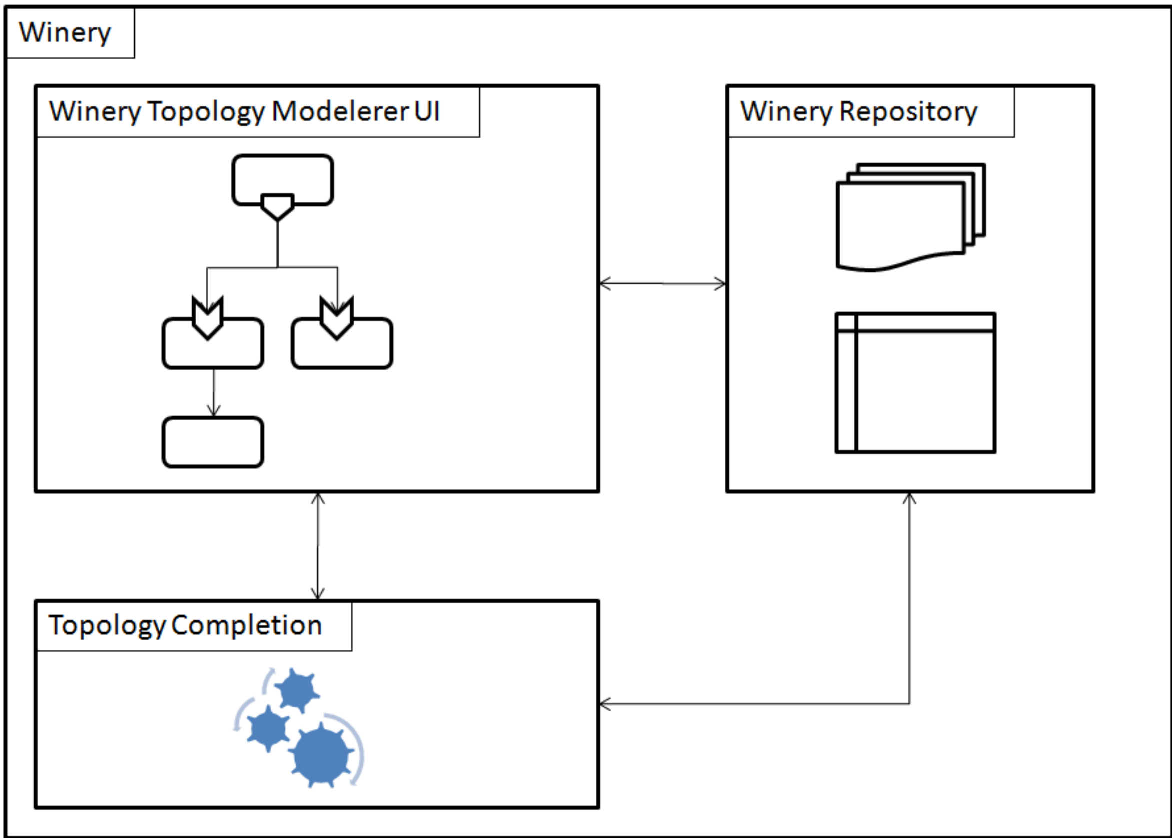

Abbildung 4.8.: Winery Integration

5. Umsetzung

Dieses Kapitel beschreibt die Umsetzung des in Kapitel 4 dargestellten Konzeptentwurfs. Die entstandene Lösung wurde in das am Institut für Architektur von Anwendungssystemen entwickelte TOSCA-Modellierungs-Tool Winery integriert und ist dort als eigenständige Komponente verfügbar. Die Implementierung dieser Komponente erfolgte in der Programmiersprache Java. Des Weiteren wurde clientseitiger Javascript-Code des Winery Topology Modelers im Rahmen dieser Arbeit angepasst und erweitert, um die Vervollständigung anzukoppeln. Für die Vervollständigung werden außerdem sowohl die Winery Repository-, als auch die Winery Common-Komponente genutzt.

Nach der Beschreibung des Entwurfs der Komponente in Kapitel 5.1, welcher Anwendungsfälle und die Architekturbeschreibung durch den UML 2-Standard darstellt, wird das Ergebnis der Implementierung anschließend in Kapitel 5.2 beschrieben. Dabei wird die Funktionalität der in Winery integrierten Komponente erläutert und mittels Screenshots visualisiert.

Nach Abschluss der Implementierung wird die Komponente in das Eclipse-Projekt Winery integriert. Da keine vollständige Testabdeckung für alle eintretenden Fälle möglich ist, werden die Tests auf Basis der in Anhang A.2 dargestellten Typen durchgeführt. Die angefertigten Testfälle sowie eintretende Fehlerfälle bzw. deren Behandlung, werden in diesem Kapitel nicht beschrieben.

Anmerkung: Bei Winery handelt es sich um ein offizielles Projekt der Eclipse-Foundation[1]. Daher wurden während der Implementierung die notwendigen Codier-Richtlinien und Lizenzbestimmung (Apache License 2.0, Eclipse Public License 1.0) der Eclipse Foundation eingehalten.

5.1. Entwurf und Architektur-Design

In diesem Kapitel wird der Entwurf der Vervollständigungs-Komponente mit Hilfe des UML 2[2]-Standards der Object Management Group[3] dargestellt.

[1] siehe http://projects.eclipse.org/projects/soa.winery
[2] Unified Modeling Language. http://www.uml.org/
[3] http://www.omg.org/

5.1.1. Anwendungsfälle

Die folgende Grafik zeigt die Anwendungsfälle, die durch die implementierten Komponenten abgedeckt werden sollen.

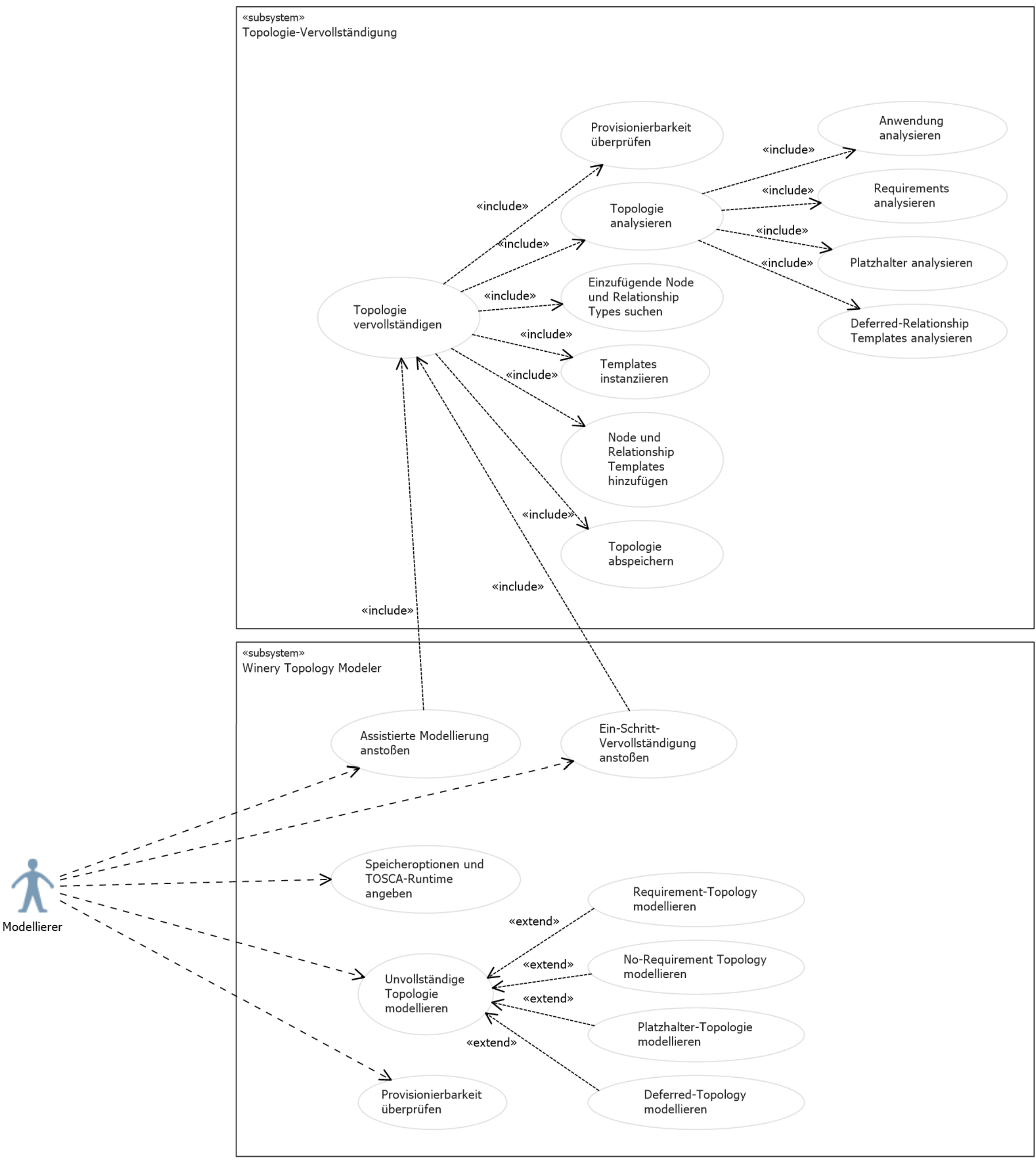

Abbildung 5.1.: Use Cases: Topologie-Vervollständigung

Wie im UML-Diagramm zu sehen ist, spalten sich die Anwendungsfälle der Topologie-Vervollständigung auf zwei Systeme auf. Dabei handelt es sich um den Winery Topology Modeler und um die Vervollständigungs-Komponente. Die dargestellten Anwendungsfälle werden im Folgenden beschrieben. Auf eine ausführliche Beschreibung, gegliedert nach UML 2-Standard, wird dabei verzichtet, da dies bereits im Konzept-Kapitel 4 abgedeckt wurde.

Die Funktionalität der geschilderten Anwendungsfälle wurde bei der anschließenden Implementierung umgesetzt (siehe Kapitel 5.2).

Anwendungsfälle des Winery Topology Modelers

- Unvollständige Topologie modellieren
  Diese Anwendungsfälle befassen sich mit der Modellierung der in Kapitel 4.3 beschriebenen unvollständigen Topologien.
  - Requirement-Topologie modellieren (Anwendungsfall 1)
  - No-Requirement-Topologie modellieren (Anwendungsfall 2)
  - Platzhalter-Topologie modellieren (Anwendungsfall 3)
  - „Deferred“-Topologie modellieren (Anwendungsfall 4)
- Topologie auf Provisionierbarkeit überprüfen
  Dieser Anwendungsfall befasst sich damit, eine Topologie auf Provisionierbarkeit zu überprüfen. Dabei wird die Provisioning-API unter Angabe der Topologie aufgerufen. Diese gibt entweder den Wert „true“, woraufhin eine Meldung angezeigt wird, dass die Topologie in dieser Form bereits provisionierbar ist oder den Wert „false“ zurück. In diesem Fall wird an der Provisioning-API für jedes Node Template abgefragt, ob es bereits provisionierbar ist. Falls das nicht der Fall ist, wird das Node Template in der Topologie markiert. Dies ist für den Nutzer ein Hinweis, dass es durch weitere Komponenten ergänzt werden muss, um provisionierbar zu sein.
- Vervollständigung anstoßen
  Hierbei wird die Schnittstelle der integrierten Vervollständigungs-Komponente aufgerufen, um die Vervollständigung anzustoßen. Dabei werden die zu vervollständigende Topologie sowie Informationen über TOSCA-Runtime und Speichermethode übergeben.
- Assistierte (Step-by-Step) Modellierung anstoßen
  Nach der Auswahl eines Node Templates kann die assistierte Modellierung für dieses angestoßen werden. Dabei werden dem Nutzer in einer gesonderten Ansicht Vorschläge gemacht, mit welchen Node bzw. Relationship Templates das ausgewählte Template verbunden werden kann. Nachdem der Modellierer eine Auswahl getroffen hat, wird diese in die Topologie eingefügt.

- Speicheroptionen und TOSCA-Runtime angeben
  Dieser Anwendungsfall ermöglicht es, die Speichermethode der Topologie nach der Vervollständigung festzulegen. Dabei gibt es die Möglichkeiten eine bestehende Topologie zu überschreiben oder eine neue Topologie anzulegen. Darüber hinaus kann festgelegt werden, ob die Topologie nach der Speicherung in einem neuen Fenster geöffnet werden soll. Unter diesen Anwendungsfall fällt außerdem das Angeben der zu verwendenden TOSCA-Runtime (bspw. OpenTOSCA [7]).

Anwendungsfälle der Vervollständigungs-Komponente

- Topologie analysieren
  Diese Anwendungsfälle dienen der Analyse von Topologien vor der Durchführung der Vervollständigung. Dabei werden Komponenten der Topologie gesucht, die auf eine notwendige Vervollständigung hinweisen. Darunter fallen Platzhalter, Requirements und Relationship Templates des Typs „Deferred“.
  - Anwendungsspezifische Komponenten analysieren
  - Requirements analysieren
  - Platzhalter analysieren
  - Relationship Templates des Typs „Deferred“ analysieren
- Topologie vervollständigen
  Um eine Topologie vervollständigen zu können, wird jeder der folgenden Anwendungsfälle benötigt:
  - Einzufügende Node Types suchen
  - Einzufügende Relationship Types suchen
  - Templates instanziieren
  - Provisionierbarkeit überprüfen
  - Topologie abspeichern
- Sonstige Anwendungsfälle (nicht im Diagramm enthalten)
  - IDs generieren
    Beim Instanziieren von Node und Relationship Templates müssen diese mit IDs eindeutig gekennzeichnet werden. Diese werden im Rahmen dieses Anwendungsfalls generiert.
  - Auf Repository zugreifen
    Während der Vervollständigung werden Node und Relationship Types gesucht, um Requirements und Platzhalter zu erfüllen. Diese stammen, wie bereits in 4.4.2 beschrieben, aus einem Repository auf welches programmatisch zugegriffen werden muss.

- Datenmodell befüllen
  Nachdem Übergabe der Topologie (als XML-Zeichenkette) an die Vervollständigungs-Komponente, wird ein Datenmodell mit allen vorhandenen Komponenten befüllt. Des Weiteren müssen aus dem Repository stammende Typen (Node Types, Relationship Types, Requirement Types etc.) in das Datenmodell übernommen werden, um diese programmatisch verarbeiten zu können.

- Nutzerabfrage durchführen
  Eine Nutzerabfrage ist notwendig sobald mehrere Relationship Types für eine Verbindung infrage kommen, mehrere Lösungs-Topologien existieren oder die Vervollständigung schrittweise bzw. assistiert durchgeführt wird. Die Kommunikation mit dem Nutzer wird in diesem Anwendungsfall abgedeckt.

5.1.2. Komponenten der Topologie-Vervollständigung

Wie aus den Anwendungsfällen abzuleiten ist, besteht die Vervollständigung aus einer Analyse-, einer Vervollständigungs- und einer Utils- Komponente. Diese Komponenten sind dabei an Winery sowie an die Provisioning-API gekoppelt. Dabei spaltet sich die Winery-Komponente in die Komponenten Winery Topology Modeler, Winery Repository und Winery Common auf, die voneinander entkoppelt sind. Die Topologie-Vervollständigungs-Komponente besitzt dabei beidseitige Abhängigkeiten zu diesen Komponenten. Des Weiteren existiert eine einseitige Abhängigkeit zur Provisioning-API. Der Aufbau der Komponenten des Gesamt-Systems ist in folgendem UML-Komponenten-Diagramm dargestellt.

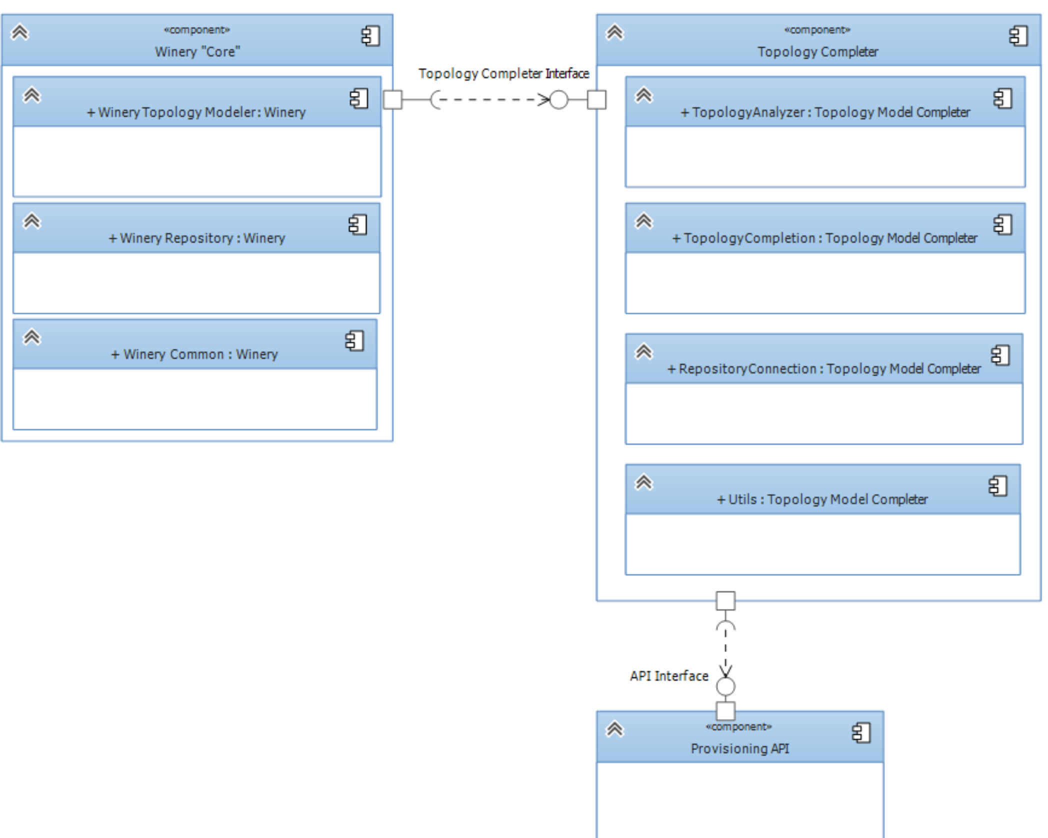

Abbildung 5.2.: Komponentenmodell der Topologie-Vervollständigung

Die Komponenten können dabei weiter in Klassen sowie den enthaltenen Methoden und Attributen untergliedert werden, welche in folgendem Klassendiagramm dargestellt sind.

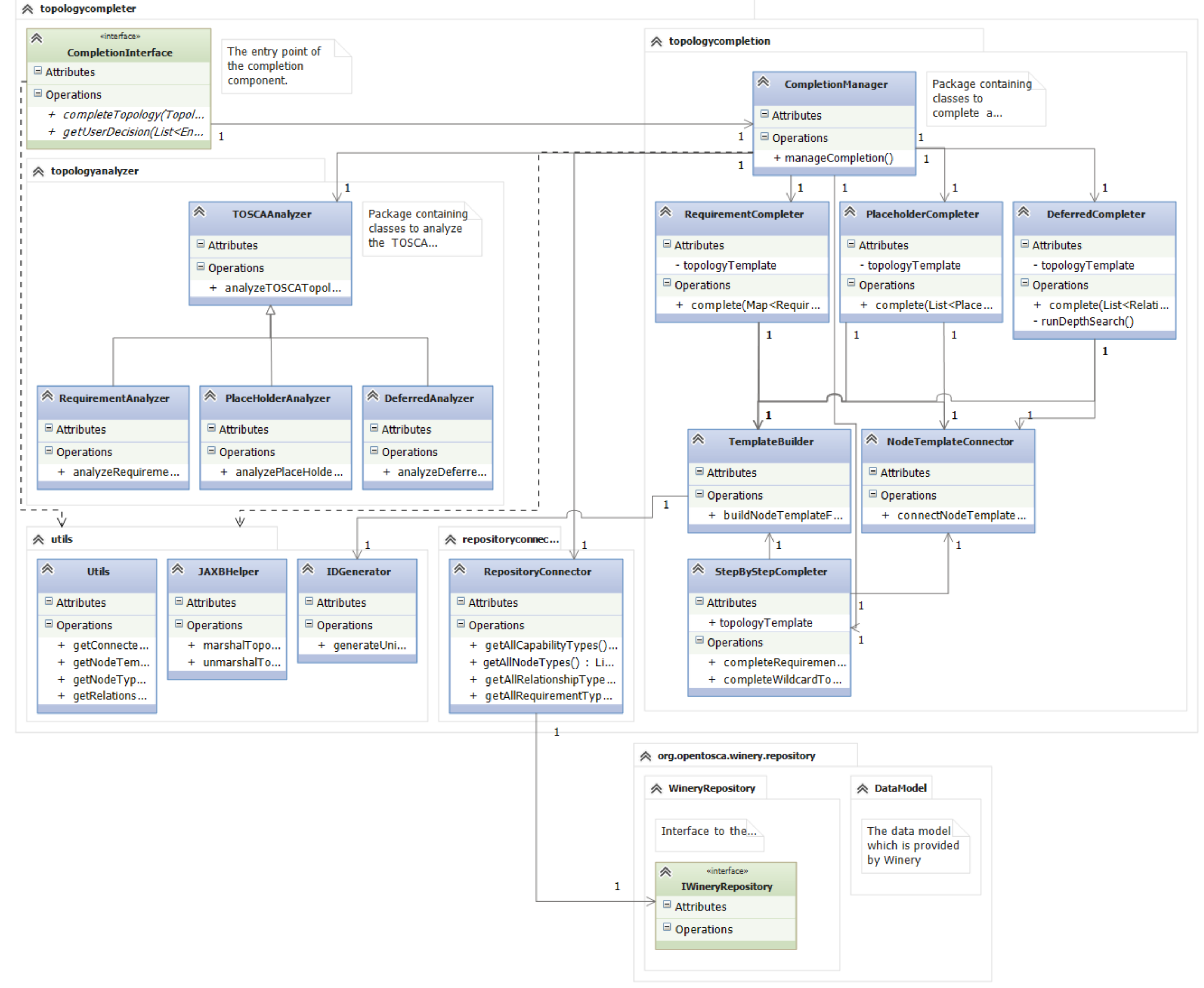

Abbildung 5.3.: Klassen der Topologie-Vervollständigung

In den folgenden Abschnitten werden die Komponenten und deren enthaltene Klassen näher beschrieben.

5.1.2.1 TopologyCompleter

Die Hauptkomponente der Vervollständigung, welche als „TopologyCompleter“ bezeichnet wird, besteht aus den Komponenten „TopologyAnalyzer“, „TopologyCompletion“, „Utils“ und „RepositoryConnector“. Die Funktionalität dieser Komponenten wird im Folgenden beschrieben.

5.1.2.2 TopologyAnalyzer-Komponente

Die Komponente „TopologyAnalyzer“ dient der Analyse von TOSCA-Topologien. Folgende Analysefunktionalität soll von dieser Komponente zur Verfügung gestellt werden.

- Analyse aller vorkommender TOSCA-Requirements
- Analyse der Platzhalter
- Analyse des Vorkommens von Relationship Templates des Typs „Deferred“

Dabei werden zum einen die, im Datenmodell abgespeicherten, Node Templates und Types der Topologie durchlaufen und auf Requirements bzw. Platzhalter analysiert, zum anderen die Relationship Templates auf den Typ „Deferred“ überprüft. Der Ablauf der Analysen wird im Folgenden beschrieben.

Requirement-Analyse
Im Rahmen der Requirement-Analyse werden alle vorhandenen Requirements einer Topologie mit den dazugehörigen Node Templates in einer Key-Value-Map abgespeichert, wobei der Schlüssel ein Requirement, der Wert ein Node Template darstellt. Dies macht es während der Vervollständigung möglich, jedes Requirement einem bestimmten Node Template zuzuordnen. Die Analyse von Requirements wird in Algorithmus 4.4 in Pseudo-Code beschrieben.

Platzhalter-Analyse
Alle Platzhalter sind, wie in 4.3.3 beschrieben, abgeleitet vom abstrakten TOSCA-Typ „Placeholder“. Um zu überprüfen ob es sich bei einem Node Template um einen Platzhalter handelt, muss dessen Typ betrachtet werden. Dieser besitzt das optionale XML-Element „DerivedFrom“, welches angibt von welchem Typ er abgeleitet wurde. Entspricht der Inhalt des Elements dem abstrakten Platzhalter-Typ „Placeholder“, kann darauf geschlossen werden, dass das Node Template einen Platzhalter und kein konkretes Template darstellt. Die gefundenen Platzhalter werden in einer Liste abgespeichert. Dabei müssen keine Informationen über die verbundenen Node Templates gespeichert werden, da diese über die Source bzw. Target-Elemente der Relationship Templates der Topologie bestimmt werden können.

„Deferred“-Relationship Template-Analyse
Bei dieser Analyse genügt es, alle Relationship Templates der Topologie zu durchlaufen und die Templates des Typs „Deferred“ in einer Liste abzuspeichern. In den XML-Elementen „SourceElement“ und „TargetElement“ ist dabei die Referenz auf die verbundenen Node Templates gespeichert. Daher müssen diese nicht gesondert untersucht werden.

5.1.2.3 Topology Completion-Komponente

Die „Topology Completion“-Komponente enthält alle Klassen und Methoden, um die Vervollständigung auszuführen. Dabei greift sie vermehrt auf die Methoden der Analyse-Komponente zu. Es herrscht also nur eine einseitige Abhängigkeit zwischen diesen Komponenten.

Die Topology Completion-Komponente enthält folgende Klassen:

- CompletionInterface
  Die Interface-Klasse „CompletionInterface“ dient als Einstiegspunkt der Vervollständigungs-Komponente und bekommt eine nicht provisionierbare Topologie nach 4.3 im XML-Format übergeben. Über diese Klasse wird außerdem die Kommunikation zwischen dem Winery Topology Modeler und der Vervollständigung geregelt. Dies ist vor allem bei Nutzerabfragen und beim Anzeigen eventuell auftretender Fehlermeldungen notwendig. Des Weiteren gibt diese Klasse die vervollständigte Topologie an den Winery Topology Modeler zurück, der diese anschließend visualisiert.
- CompletionManager
  Diese Klasse steuert die Vervollständigung, indem zu einer Topologie passende Methoden ausgewählt und aufgerufen werden. Nach dem Aufruf der Analyse-Komponente werden die benötigten Methoden-Aufrufe, abhängig von den Analyse-Ergebnissen, zusammengestellt. Somit kann die Vorgehensweise je nach eintretendem Anwendungsfall angepasst werden.
- RequirementCompleter
  Erfüllt die Requirements einer Topologie durch das Einfügen passender Node Templates. Dabei wird das Repository aufgerufen, welches alle vorhandenen Node Types zurückgibt. Anschließend wird für jeden erhaltenen Node Type überprüft, ob dieser über eine zum Requirement passende Capability verfügt. Ist dies der Fall, wird die Klasse „TemplateBuilder“ aufgerufen, welche ein Node Template aus diesem Typ instanziiert. Daraufhin wird die Klasse „NodeTemplateConnector“ aufgerufen, um ein oder mehrere passende Relationship Templates für eine Verbindung zum instanziierten Node Template zu erhalten. Die Templates werden anschließend in die Topologie eingefügt und das Requirement als erfüllt vermerkt. Bei mehreren möglichen einzufügenden Templates werden Topologie-Kopien erstellt oder es wird eine Nutzerabfrage über das „CompletionInterface“ durchgeführt.
- PlaceholderCompleter
  Diese Klasse dient dem Erfüllen von Platzhaltern der Topologie. Dabei erhält sie eine

Liste von Platzhalter-Node Templates aus der Analyse-Komponente. Für jeden dieser Platzhalter wird über dessen Typ ein passender Node Type im Repository gesucht. Aus diesem wird mit Hilfe der „TemplateBuilder"-Klasse ein Node Template instanziiert und der Topologie hinzugefügt. Analog zur „RequirementCompleter"-Klasse wird anschließend ein passendes Relationship Template eingefügt. Der Platzhalter sowie dessen generische Verbindungen werden daraufhin aus der Topologie entfernt.

- DeferredCompleter
  Die „DeferredCompleter"-Klasse vervollständigt Topologien, die ein oder mehrere Relationship Templates des Typs „Deferred" enthalten. Dabei wird der Algorithmus aus 4.4.3.7 verwendet.
- StepByStepCompleter
  Diese Klasse steuert die schrittweise bzw. assistierte Vervollständigung einer Topologie. Dabei werden einzufügende Node und Relationship Templates gesucht und dem Nutzer in jedem Schritt über das CompletionInterface zur Auswahl gegeben. Nachdem eine Auswahl getroffen wurde, wird die Vervollständigung fortgesetzt. Dies wiederholt sich bis eine provisionierbare Topologie entstanden ist.
- TemplateBuilder
  Die Klasse „TemplateBuilder" dient der Instanziierung von Node oder Relationship Templates. Sie bekommt hierfür ein Objekt des Typs „EntityType" übergeben, von dem sowohl Relationship Types als auch Node Types ableiten. Abhängig vom Typ des erhaltenen Objekts, legt diese Klasse ein neues Node oder Relationship Template an. Die ID des angelegten Templates wird dabei durch die Klasse „IDGenerator" erzeugt. Enthalten die Typen Default-Eigenschaften, werden diese den Templates hinzugefügt.
- NodeTemplateConnector
  Diese Klasse verbindet Node Templates durch passende Relationship Templates. Die für die Instanziierung verwendeten Relationship Types können dabei durch Algorithmus 4.1 gefunden werden. Aus diesen werden daraufhin über die „TemplateBuilder"-Klasse Relationship Templates instanziiert und dem Nutzer über das „CompletionInterface" vorgeschlagen. Die getroffene Auswahl wird anschließend zurückgegeben.

5.1.2.4 RepositoryConnector-Komponente

Die Komponente „RepositoryConnector" dient dem programmatischen Zugriff auf das Winery Repository um Node Types, Relationship Types, Requirement Types und andere dort abgespeicherte TOSCA-Elemente zu erhalten. Neben dem lesenden Zugriff dieser Komponente existiert darüber hinaus ein schreibender Zugriff auf das Repository zum Abspeichern der vervollständigten Topologie. Die Interaktion mit dem Repository geschieht dabei ausschließlich über die angebotene REST-Schnittstelle mittels HTTP (Hypertext Transfer Protocol)-Aufrufen.

5.1.2.5 Utils-Komponente

Neben den Hauptkomponenten wurde eine Utils-Komponente geschaffen, die über diverse, bei der Vervollständigung von TOSCA-Topologien verwendete, Hilfsmethoden verfügt.

- IDGenerator
  Die Klasse „IDGenerator“ generiert zufällige UUIDs (Universally Unique Identifier) für instanziierte Templates.
- Suchen von Node Templates nach ID
  Diese Methode nutzt die als JAXB-Objekt abgespeicherte TOSCA-Topologie, um über die enthaltenen Node Templates zu iterieren und ein Template zu einer gegebenen ID zurückzugeben.
- Suchen von Relationship Templates nach ID
  Analog zur Suche nach Node Templates.
- JAXB-Marshaller und Unmarshaller-Methoden
  Diese Methoden sind notwendig, da die Topologie als XML-Zeichenkette der Schnittstelle übergeben wird und diese mittels JAXB-Unmarshaller in ein TOSCA-Java-Objekt umgewandelt werden muss. Dieses Objekt wird nach der Vervollständigung durch einen JAXB-Marshaller erneut in eine XML-Zeichenkette umgewandelt, um die Topologie per REST-Aufruf im Repository abspeichern zu können.
- TopologyBuilder um XML zu bearbeiten (nicht im Diagramm enthalten)
  Da Winery für die Position (X- und Y-Koordinaten) der Node Templates im grafischen Editor und für eigene Namespaces weitere XML-Attribute benötigt, werden diese nach der Vervollständigung mittels JAXB durch die TopologyBuilder-Klasse eingefügt.

5.1.3. Sequenz Diagramme zur Topologie-Vervollständigung

In diesem Abschnitt werden bestimmte Abläufe mit Sequenz-Diagrammen dargestellt, um die Implementierung besser nachvollziehen zu können.

Der Ablauf der Vervollständigung ohne Nutzerinteraktion wird in folgendem Sequenz-Diagramm beschrieben. Dabei ist die Lösungs-Topologie eindeutig, sodass keine Auswahl aus mehreren Topologien notwendig ist.

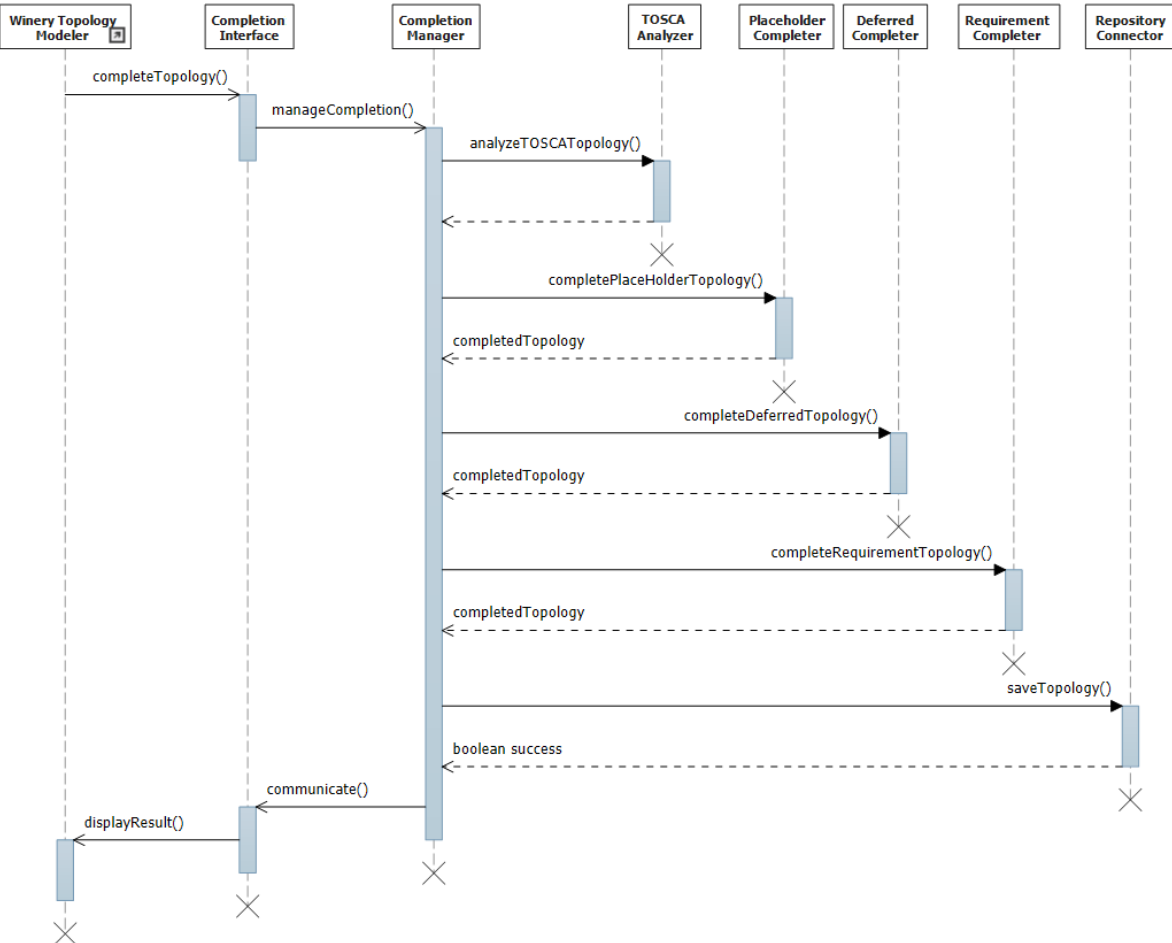

Abbildung 5.4.: Ablauf der Topologie-Vervollständigung

Wie in diesem Sequenz-Diagramm zu sehen ist, gehen alle Aufrufe von der Completion Manager-Klasse aus, die die Steuerung der Vervollständigung übernimmt. Nach dem Aufruf der Topologie-Analyse wird die Vervollständigung über die Klasse TopologyCompleter angestoßen. Diese nutzt nach erfolgreicher Vervollständigung die Klasse RepositoryConnector, um die Topologie im Repository abzuspeichern. Anschließend wird diese im Winery Topology Modeler visualisiert.

Das folgende Sequenz-Diagramm zeigt den Ablauf einer Vervollständigung, die eine Nutzerinteraktion benötigt. Dabei wurden die Aufrufe der Vervollständigung durch das Objekt „Completer“ abstrahiert, da diese analog zum vorigen Diagramm aus Abbildung 5.4 durchgeführt werden.

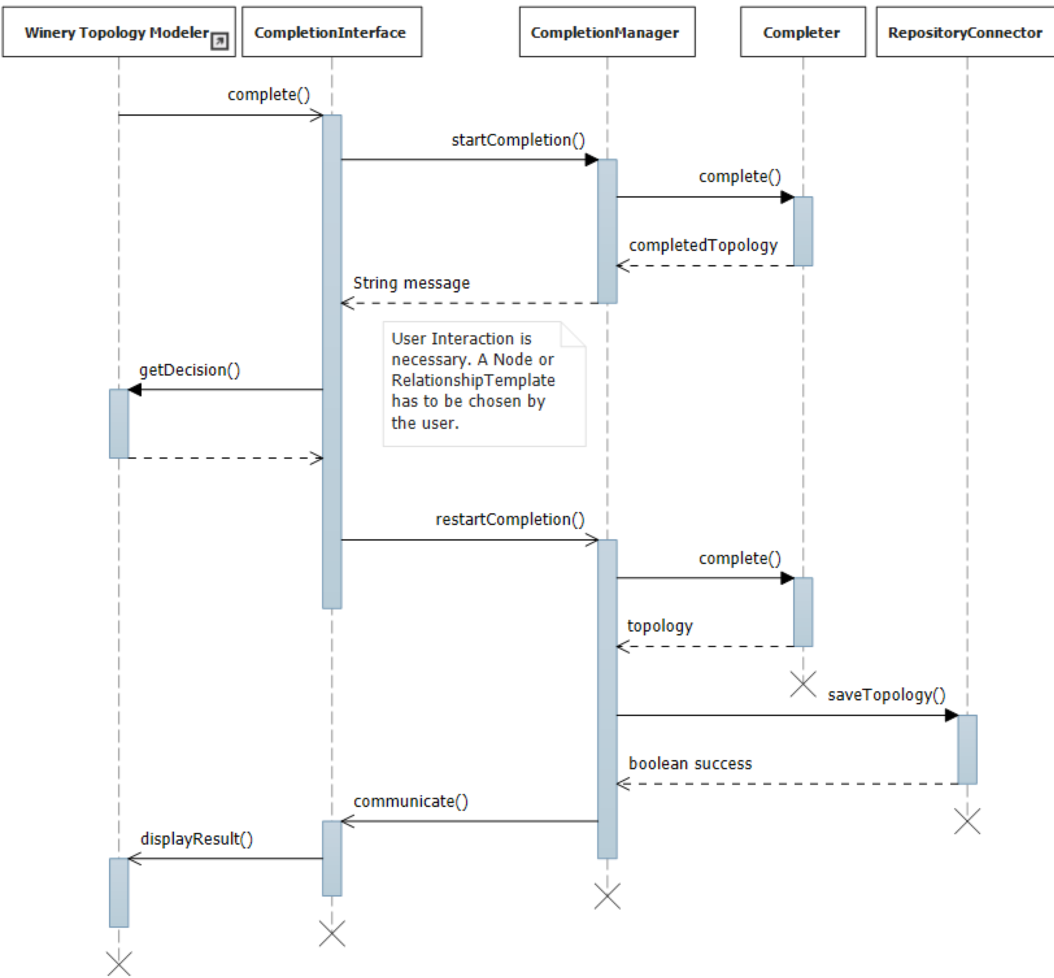

Abbildung 5.5.: Ablauf der Topologie-Vervollständigung mit Nutzerinteraktion

Im Gegensatz zur Vervollständigung ohne Nutzer-Interaktion wird die Topologie-Vervollständigung unterbrochen, sobald eine Nutzer-Entscheidung notwendig ist. Dabei werden die Topologie und eine Liste der auszuwählenden Templates an den Winery Topology Modeler übergeben, welcher diese visualisiert. Sobald der Nutzer eine Auswahl getroffen hat, wird die Vervollständigung an der unterbrochenen Stelle fortgesetzt.

5.1.4. Winery Topology Modeler-Erweiterung

Um einen Anschluss an die Vervollständigungs-Komponente zu schaffen, muss der Topology Modeler von Winery angepasst und erweitert werden. Hierzu ist es notwendig einen weiteren Button im Winery Topology Modeler zu implementieren, der die Vervollständigung serverseitig anstößt. Des Weiteren muss die Logik für die Nutzerabfrage sowie die assistierte Modellierung in Winery implementiert werden. Da der Winery Topology Modeler auf Java Server Pages (JSP) [19] basiert, wurde diese Technologie auch für die Winery-Erweiterung verwendet. Abbildung 5.6 zeigt eine Übersicht der Komponenten von Winery mitsamt der integrierten Vervollständigungs-Komponente.

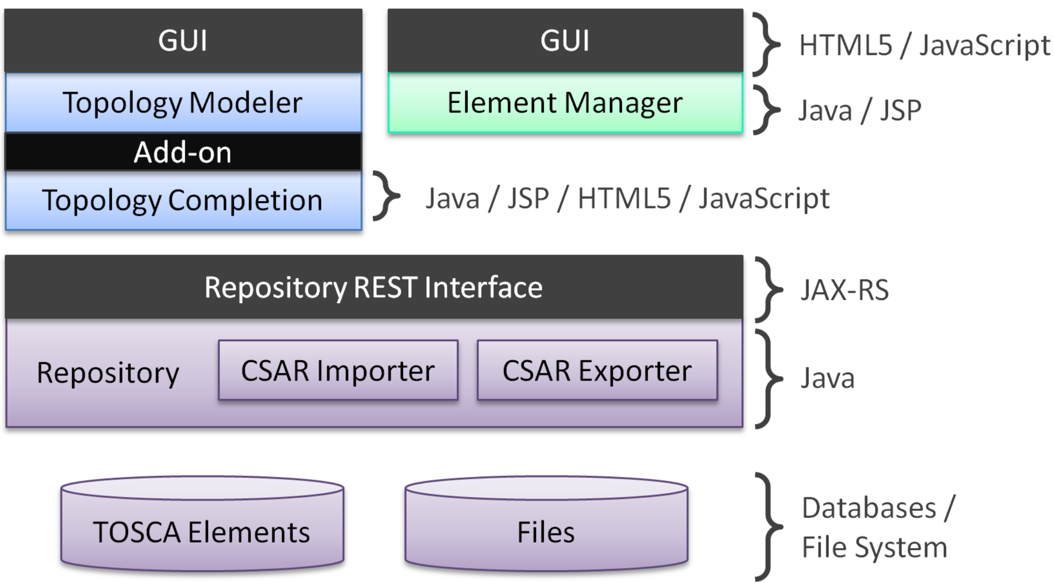

Abbildung 5.6.: Winery Komponenten-Übersicht (auf Basis von [2])

5.1.4.1 Erweiterung bestehender Winery-Komponenten

Bei der Anbindung der Vervollständigung muss lediglich die Haupt-Komponente (index.jsp) des Winery Topology Modelers angepasst werden. Diese beschreibt mittels JSP die Oberfläche des Winery Topology Modelers und führt Methodenaufrufe bei der Interaktion mit dieser aus. Dabei werden der Oberfläche die Buttons „Complete Topology“ und „Check Provisionability“ hinzugefügt, welche mit Event Handlern versehen werden. Bei der Auswahl einer der Buttons wird dabei ein Event ausgelöst, welches zu einem Methoden-Aufruf führt. Über diesen kann die Vervollständigung serverseitig mittels AJAX[4]-Request angestoßen werden. Des Weiteren wird in dieser Komponente ein Dialog implementiert, in dem der Nutzer Informationen über Speichermethode und TOSCA-Runtime angeben kann (siehe Abbildung 5.9).

[4] Asynchronous JavaScript and XML

5.1.4.2 Neu geschaffene Winery-Komponenten (JSPs)

Neben der Erweiterung der Winery-Haupt-Komponente werden außerdem neue Komponenten mittels JSP geschaffen, um die - in Java geschriebene - serverseitige Vervollständigungs-Komponente anzukoppeln. Die neuen Komponenten dienen dabei der Visualisierung von Lösungs-Topologien, der Nutzerabfragen sowie des Ergebnisses der Überprüfung auf Provisionierbarkeit.

Auswahl von Lösungs-Topologien
Existieren nach der Vervollständigung mehrere Lösungs-Topologien, werden diese in einer gesonderten Ansicht angezeigt. Hierfür wird eine neue Komponente „TopologyTemplate-Renderer“ eingeführt, welche vollständige TOSCA-Topologien visualisiert. Dabei wird der bereits implementierte „NodeTemplateRenderer“ von Winery genutzt. Dieser durchläuft alle in der Topologie enthaltenen Node Templates und rendert diese zu grafischen Objekten. Anschließend werden diese grafisch visualisierten Node Templates mit Hilfe der Javascript-Bibliothek "jsPlumb"[5] - auf Basis der Relationship Templates der Topologie - verbunden.

Nutzer-Abfragen
Eine Nutzer-Abfrage zur Auswahl einzufügender Templates ist notwendig, sobald die Topologie Schritt-für-Schritt vervollständigt werden soll oder wenn mehrere Relationship Templates für eine Verbindung in Frage kommen. Auch hier wird die eingeführte „TopologyTemplateRenderer“-Komponente genutzt, um die Auswahl zu visualisieren. Eine rein textuelle Auswahl würde die Nutzbarkeit der Lösung ansonsten stark einschränken.

Die Abhängigkeiten der beschriebenen Komponenten werden in folgender Abbildung dargestellt. Dabei werden nur die für die Vervollständigung notwendigen Komponenten beschrieben.

[5] http://jsplumbtoolkit.com/doc/home

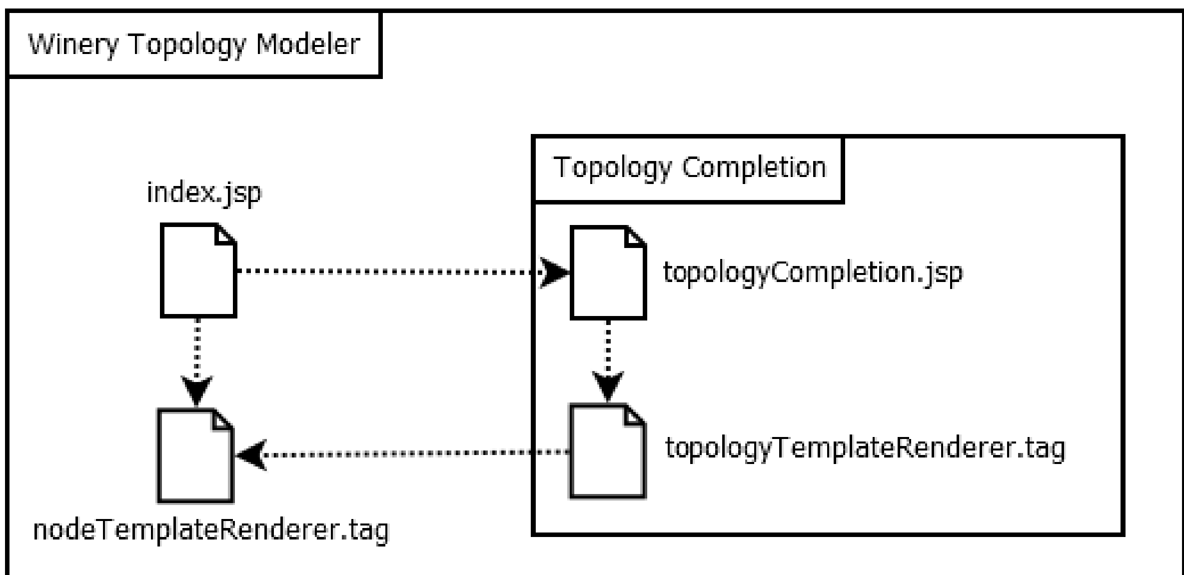

Abbildung 5.7.: Abhängigkeiten des Winery Topology Modelers

Wie in der Abbildung zu sehen ist, greift die index.jsp des Winery Topology Modelers auf die topologyCompletion.jsp zu. Dies geschieht sobald der, in der index.jsp implementierte, Button zum Anstoßen der Vervollständigung ausgewählt wird. Die topologyCompletion.jsp ruft die serverseitig implementierten Algorithmen der Vervollständigung anschließend auf und verarbeitet die Antwort. Sind Nutzerabfragen notwendig wird das topologyTemplateRenderer Tag verwendet, um die Auswahl von Node und RelationshipTemplates oder ganzen Lösungstopologien zu visualisieren. Dabei nutzt das topologyTemplateRenderer Tag den bereits implementierten nodeTemplateRenderer, um die NodeTemplates zu grafischen Objekten zu rendern. Dieser wurde bisher von der index.jsp genutzt, um Node Templates im Editor darzustellen.

5.2. Implementierung und Ergebnis

In diesem Kapitel wird das Ergebnis der Implementierung der Topologie-Vervollständigung auf Basis des vorigen Entwurfs - integriert in den Winery Topology Modeler - beschrieben. Die Implementierung wurde clientseitig mittels Javascript, serverseitig mittels Java umgesetzt. Dabei wurden Java Server Pages (JSP) für die Generierung der Benutzeroberfläche verwendet bzw. bestehende Winery-JSPs erweitert. Das TOSCA-Datenmodell von Winery wurde mit Hilfe der „Java Architecture for XML Binding" (JAXB) über das TOSCA-XML-Schema automatisch generiert. Dies stellt sicher, dass jederzeit auf einem validen, d.h. schemakonformen TOSCA-XML-Dokument gearbeitet wird.

Die Funktionalität der entstandenen Komponente lässt sich wie folgt aufteilen:

1. Vervollständigung einer Topologie in einem Schritt nach den Anwendungsfällen aus Kapitel 4
2. Schrittweise Vervollständigung einer Topologie (Assistierte Modellierung)
3. Überprüfung einer Topologie auf Provisionierbarkeit
4. Suchen eines Node Template-Containers

In den nachfolgenden Kapiteln wird erläutert, wie diese Funktionalität umgesetzt wurde und das End-Resultat dargestellt.

5.2.1. Vervollständigung einer Topologie in einem Schritt

Der genaue Ablauf der Vervollständigung wurde bereits in den Kapiteln 4.4 und 5.1 beschrieben und wird daher im Folgenden nicht erneut geschildert.

Um eine Topologie „in einem Schritt" zu vervollständigen werden folgende Schritte im Winery Topology Modeler ausgeführt:

Zur Durchführung der Vervollständigung fertigt der Modellierer zuerst eine unvollständige bzw. nicht provisionierbare TOSCA-Topologie nach einem der Anwendungsfälle aus Kapitel 4.3 an (siehe Abbildung 5.8).

Das Beispiel aus Abbildung 5.8 verwendet eine Kombination der Anwendungsfälle 3 und 4. Durch die Verwendung einer „Deferred-Verbindung" (rot markiert) wird der Cloud-Provider Amazon festgelegt und die dazwischenliegenden Komponenten offen gelassen. Durch den Datenbank-Platzhalter kann sichergestellt werden, dass das bereits konkret modellierte Datenbank-Management-System sowie das Betriebssystem „Ubuntu" verwendet werden kann.

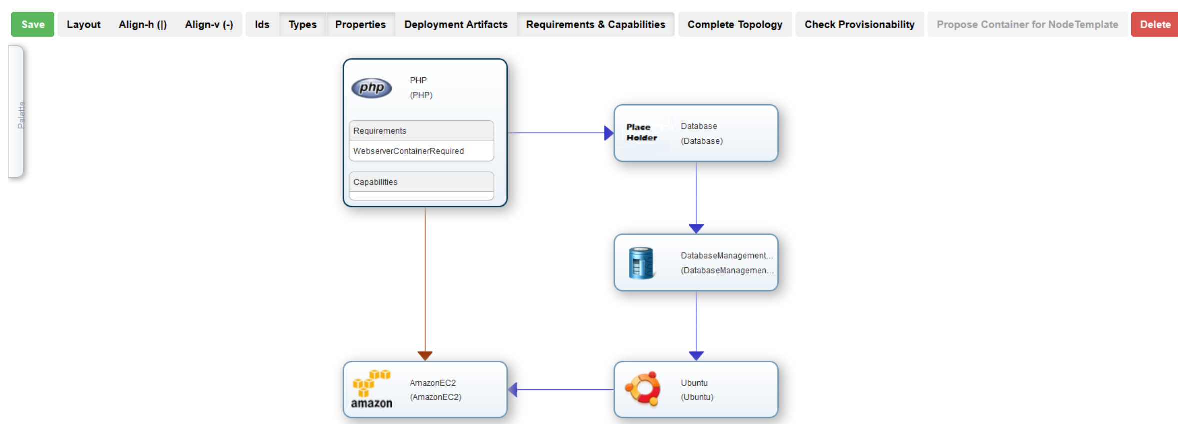

Abbildung 5.8.: Modellierung einer unvollständigen TOSCA-Topologie

Nach der Modellierung einer Topologie wählt der Modellierer „Complete Topology" aus. Daraufhin öffnet sich ein Dialog, in dem sowohl Informationen über die Speicherung der Topologie, als auch die zu verwendende TOSCA-Runtime abgefragt werden. Dabei kann festgelegt werden, ob die Topologie überschrieben oder ob eine neue Topologie im Repository angelegt werden soll. Soll eine neue Topologie angelegt werden, ist ein Name und ein valider Namespace für die eindeutige Speicherung anzugeben. In diesem Dialog kann auch die verwendete TOSCA-Runtime ausgewählt werden. Momentan steht hierfür nur die TOSCA-Runtime OpenTOSCA zur Verfügung. In Zukunft sollen jedoch auch weitere TOSCA-Laufzeitumgebungen unterstützt werden. Das Feld „Complete topology Step-by-Step" wird bei der Vervollständigung in einem Schritt nicht ausgewählt. Die folgende Grafik zeigt den eben beschriebenen Auswahl-Dialog.

Abbildung 5.9.: Topologie-Vervollständigungs-Dialog

Nach dem Befüllen des Dialogs wird die Topologie-Vervollständigung nach Auswahl von „Complete Topology“ durch einen serverseitigen Aufruf der Vervollständigungs-Komponente angestoßen. Dabei wird die grafisch modellierte Topologie in eine TOSCA-XML-Zeichenkette (engl.: String) umgewandelt und übergeben. Außerdem werden die Repository-URL sowie die ausgewählten Speicher-Optionen übergeben. Der Topologie-XML-String wird daraufhin in das JAXB-Datenmodell, basierend auf dem TOSCA-XML-Schema Version 1.0, geladen. Daraufhin wird die Analyse-Phase der Topologie-Vervollständigung ausgeführt, in der die Topologie auf existierende Requirements, Platzhalter und Relationship Templates des Typs „Deferred“ überprüft wird. Existieren keine dieser Elemente, kann die Topologie als vollständig definiert werden. Ist mindestens eines dieser Elemente vorhanden, wird die Vervollständigung angestoßen. Das Steuern der Vervollständigung wird dabei durch die Klasse „CompletionManager“ übernommen. Diese ruft, abhängig vom Anwendungsfall aus 4.3, passende Methoden zur Vervollständigung auf. Sind sowohl Requirements, Platzhalter als auch Relationship Templates des Typs „Deferred“ vorhanden, wird die Vervollständigung in folgender Reihenfolge durchgeführt:

1. Vervollständigung der Platzhalter
   Da Platzhalter bei der Vervollständigung der anderen Anwendungsfälle nicht mehr vorhanden sein dürfen, werden diese zu Beginn erfüllt.

2. Vervollständigung der Relationship Templates des Typs „Deferred“

3. Vervollständigung von Requirements

Nachdem die Algorithmen der Vervollständigung (siehe Kapitel 4.4.3) abgeschlossen sind, werden die vervollständigten Topologien (meist ist die Lösung nicht eindeutig) an Winery übergeben und dem Nutzer zur Auswahl angezeigt. Aus diesen kann er anschließend eine oder mehrere Topologien auswählen, die in das Repository übernommen werden sollen. Dabei kann für jede Topologie-Alternative festgelegt werden, ob bzw. auf welche Art diese abgespeichert wird. Existiert eine eindeutige Lösung, erscheint kein Auswahl-Dialog und es werden die Speicheroptionen aus dem Vervollständigungs-Dialog (Abbildung 5.9) übernommen. Nach der Auswahl einer oder mehrerer Topologien werden sie mittels REST-Aufruf PUT oder POST (beim Anlegen einer neuen Topologie) im Winery Repository abgespeichert und im Winery Topology Modeler angezeigt. Hat der Nutzer im Vervollständigungs-Dialog „Open topology in new window“ ausgewählt, öffnet sich ein neues Browser-Fenster, in dem die Topologie angezeigt wird.

Abbildung 5.10 zeigt den Auswahl-Dialog der Lösungs-Topologien.

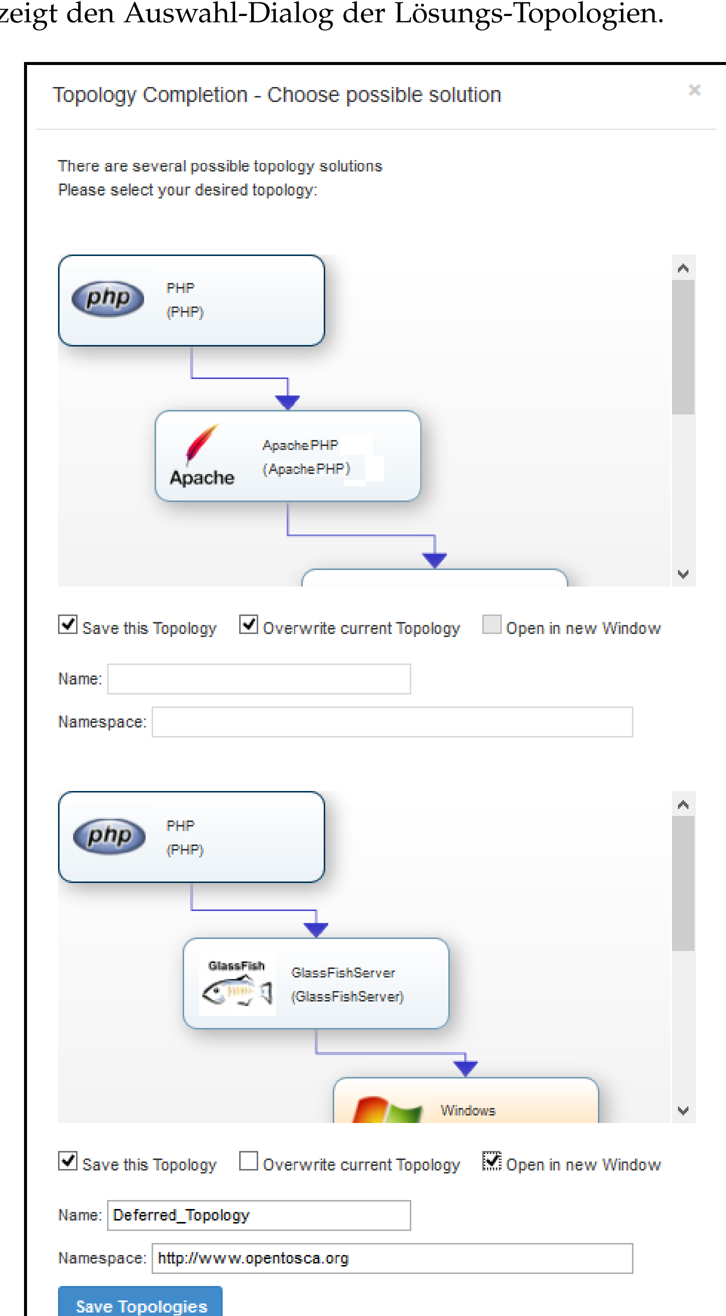

Abbildung 5.10.: Dialog zur Auswahl einer Topologie-Lösung

Nach der Auswahl wird die vervollständigte Topologie im Winery Topology Modeler angezeigt (siehe folgende Abbildung). Dabei wurden die bei der Modellierung (siehe Abbildung 5.8) offen gelassenen Infrastruktur-Komponenten ergänzt, sodass eine vollständige, auf der OpenTOSCA-Laufzeitumgebung provisionierbare Topologie entstanden ist.

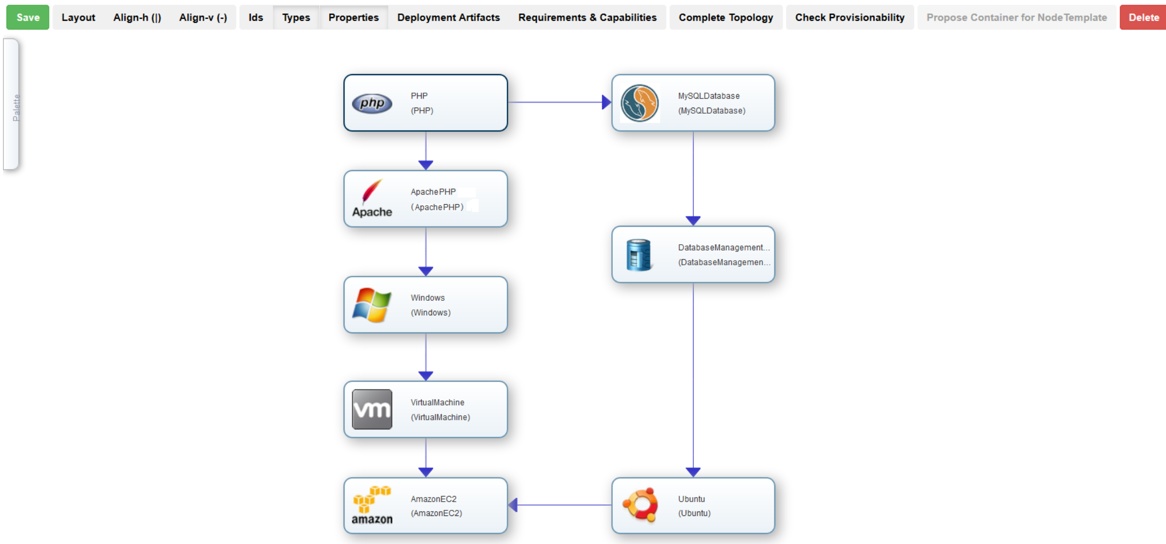

Abbildung 5.11.: End-Resultat der Topologie-Vervollständigung

Nutzer-Interaktion bei der Vervollständigung in einem Schritt

Obwohl die Vervollständigung „in einem Schritt“ erfolgt, kann eine Nutzerinteraktion während dem Ablauf der Vervollständigung notwendig sein. Für die Verbindung zweier Node Templates kommen oftmals mehrere Relationship Templates in Frage. Die Auswahl der einzufügenden Relationship Templates kann dabei nicht automatisch erfolgen, da dies sonst zu unerwünschten Ergebnissen führen kann. Aufgrund dessen, dass eine Abfrage der einzufügenden Relationship Templates am Ende der Vervollständigung oder das Einfügen aller möglichen Relationship Templates der Übersicht schadet, wird diese Abfrage an auftretenden Entscheidungspunkten während der Vervollständigung durchgeführt. Dabei öffnet sich ein Dialog, in dem der Nutzer ein oder mehrere Relationship Templates auswählen kann, die für die Verbindung zweier Node Templates verwendet werden. Nach der Auswahl wird die Topologie-Vervollständigung fortgesetzt. Abbildung 5.12 zeigt den beschriebenen Auswahl-Dialog.

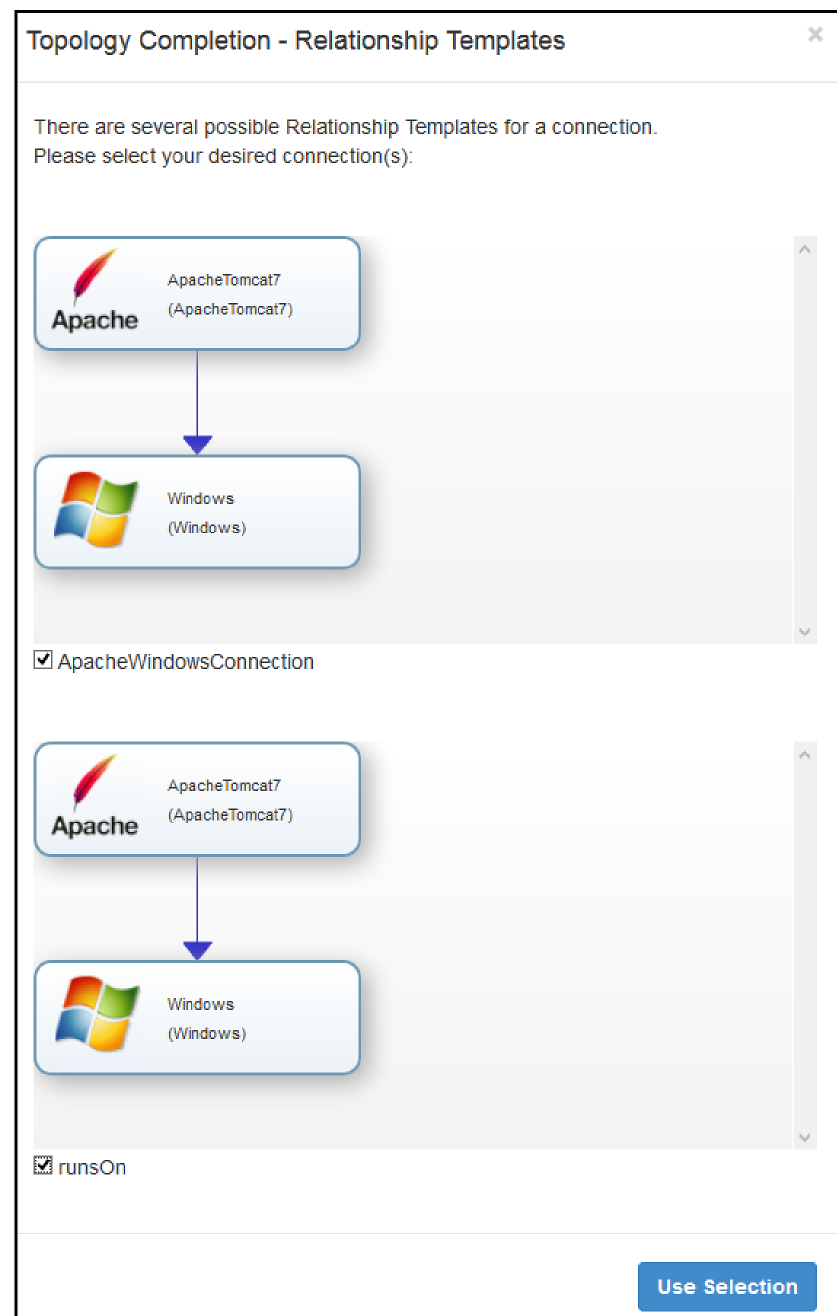

Abbildung 5.12.: Dialog zur Auswahl von Relationship Templates

5.2.2. Schrittweise Vervollständigung einer Topologie / Assistierte Modellierung

Bei der schrittweisen Vervollständigung wird die Entscheidung über zu ergänzende Node bzw. Relationship Templates in jedem Schritt vom Nutzer abgefragt. Dadurch erübrigt sich die Auswahl der Lösungs-Topologien nach der Vervollständigung sowie die Verwendung von Platzhaltern. Nach der Modellierung der unvollständigen Topologie wählt der Nutzer analog zur Ein-Schritt-Vervollständigung „Complete Topology" aus, woraufhin sich die Nutzerabfrage (siehe Abbildung 5.9) öffnet. In dieser wählt der Nutzer „Complete topology Step-by-Step" aus, um die Vervollständigung schrittweise durchzuführen. Nach Auswahl von „Complete Topology" wird die Vervollständigung angestoßen.

Im Gegensatz zum Ein-Schritt-Ansatz wird dem Nutzer bei jeder Entscheidung über eingefügte Node oder Relationship Templates eine Auswahl-Ansicht angezeigt. Somit kann der Nutzer Schritt für Schritt mit Hilfe eines „Wizards" durch die Topologie-Vervollständigung navigieren. Dies hat den Vorteil, dass er zum einen die Vervollständigung besser nachvollziehen kann, zum anderen eine aufwändige Auswahl aus mehreren Lösungs-Topologien entfällt. Des Weiteren kann der Topologie-Modellierer die schrittweise Vervollständigung durch die Auswahl von „Cancel Automatic Completion" an beliebiger Stelle unterbrechen, um selbst weitere Komponenten hinzuzufügen.

Nach dem letzten ausgewählten Node Template wird die vervollständigte Topologie per REST-Aufruf im Repository abgespeichert. In Abbildung 5.13 ist ein Auswahl-Dialog der schrittweisen Vervollständigung zu sehen. Dabei kommt für eine Verbindung zum Node Template „GlassFishServer" nur ein Relationship Template in Frage, für das Template „ApachePHP" können zwei Relationship Templates eingefügt werden. Sollte sich der Nutzer für das Template „ApachePHP" entscheiden, kann er auswählen, ob eines oder beide der möglichen Relationship Templates in die Topologie übernommen werden sollen.

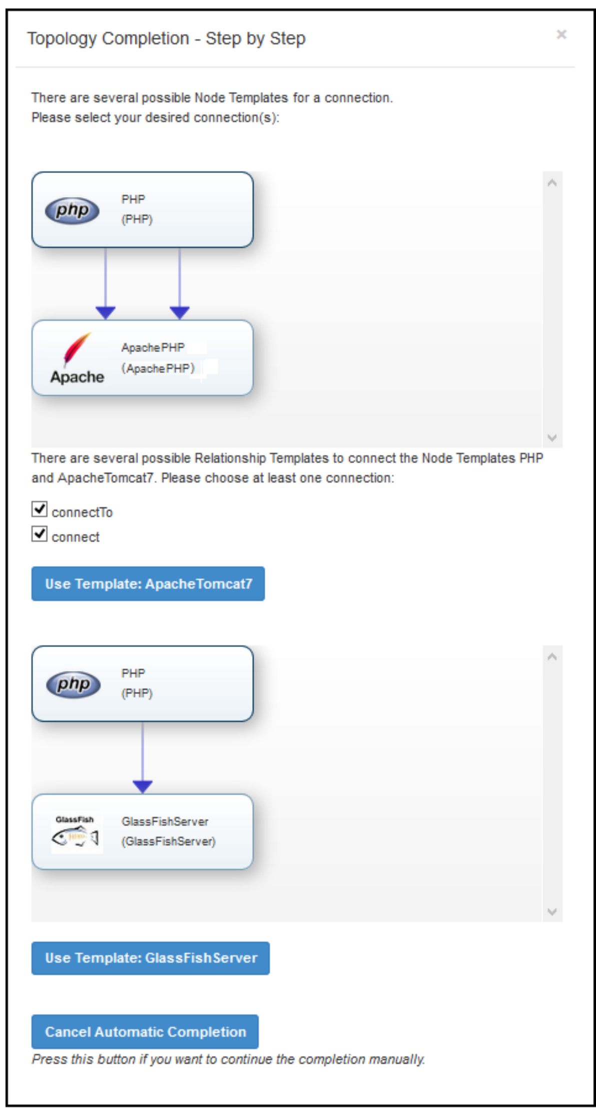

Abbildung 5.13.: Auswahl von Node und Relationship Templates beim Step-by-Step-Ansatz

5.2.3. Überprüfen einer Topologie auf Provisionierbarkeit

Neben der Vervollständigung wurde eine Überprüfung auf Provisionierbarkeit einer Topologie implementiert. Hierbei wird die Provisioning-API (siehe 4.4.3.3) aufgerufen, welche Informationen über die Provisionierbarkeit einer Topologie oder einzelner Node Templates liefern kann. Bei der Auswahl von „Check Provisionability" wird zuerst die gesamte Topologie auf Provisionierbarkeit überprüft. Ist diese nicht provisionierbar, werden die einzelnen Node Templates überprüft. Stellt sich dabei heraus, dass einzelne Node Templates nicht provisionierbar sind, werden diese rot markiert. Dies ist für den Modellierer der Hinweis, dass diese ergänzt werden müssen. In Abbildung 5.14 ist die Überprüfung auf Provisionierbarkeit dargestellt. Dabei wurde das Node Template „PHP" überprüft, welches aufgrund fehlender Elemente so noch nicht auf der gewählten TOSCA-Laufzeitumgebung provisioniert werden kann. Dies wird durch die rote Umrandung des Node Templates dargestellt.

Abbildung 5.14.: Provisionierbarkeits-Überprüfung

5.2.4. Suchen eines Container-Node Templates

Die Suche nach einem Container knüpft direkt an die Überprüfung einer Topologie auf Provisionierbarkeit an. Nachdem nicht provisionierbare Node Templates markiert wurden, kann die Suche auf diesen Elementen durchgeführt werden. Dazu wählt der Benutzer zuerst ein markiertes Node Template und anschließend „Propose Container Node Template" aus, um eine Auswahl an Containern zu erhalten, auf denen das ausgewählte Node Template provisioniert werden kann. In dieser Auswahl-Ansicht wählt der Benutzer einen Container, der anschließend in die Topologie eingefügt wird. Daraufhin wird die Topologie erneut auf Provisionierbarkeit überprüft und es werden gegebenenfalls nicht provisionierbare Node Templates markiert. Für diese kann die Suche erneut durchgeführt werden. Abbildung 5.15 zeigt die Auswahl-Ansicht der Container.

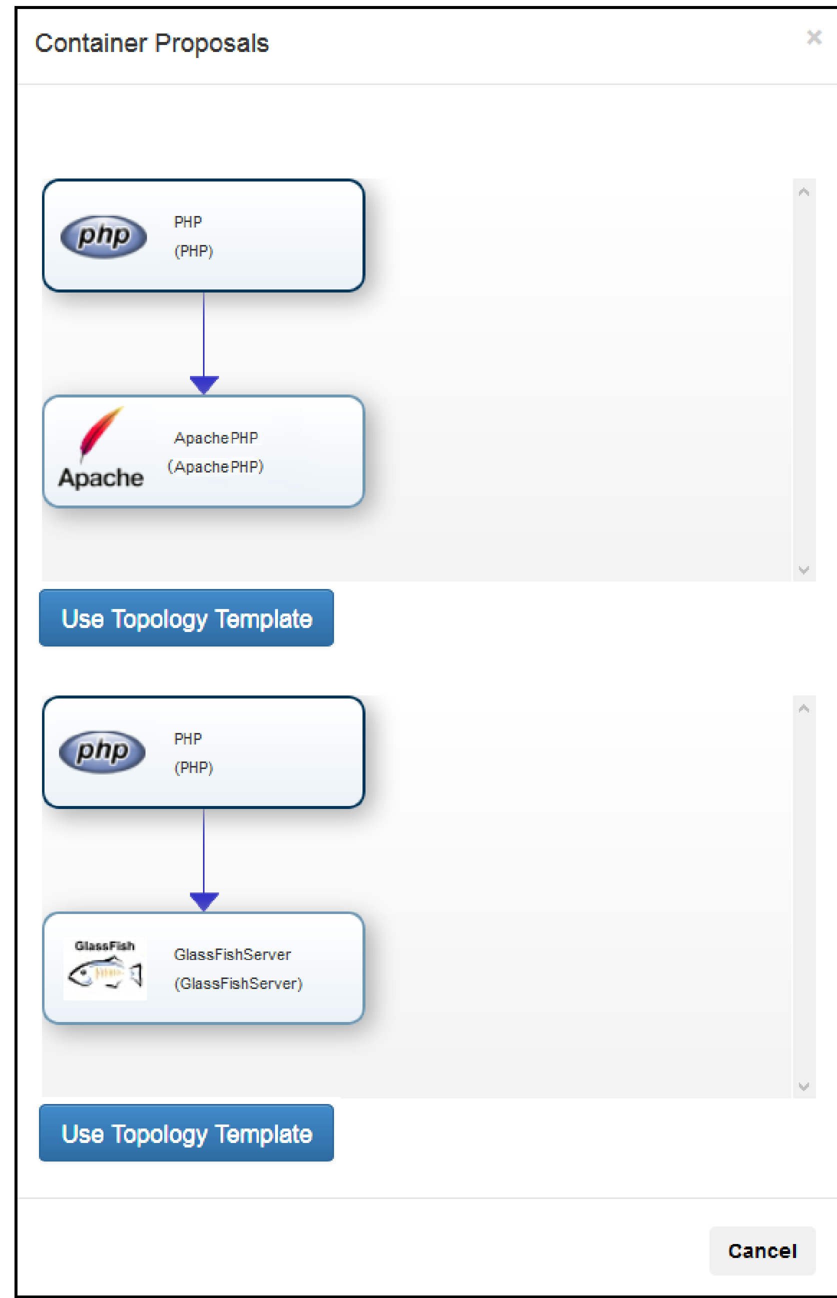

Abbildung 5.15.: Auswahl eines Containers

Zusatz-Bemerkung: Typ-Umfang
Um die Vervollständigung zu testen, wurde ein konkreter Typ-Umfang (Node Types, Relationship Types, Requirements, Capabilities) festgelegt. Die bei der Implementierung zum Testen der Funktionalität verwendeten Typen sind den Abbildungen A.1 und A.2 zu entnehmen.

5.2.5. Sonderfälle

In diesem Kapitel werden Sonderfälle, die während der Vervollständigung von Topologien auftreten können sowie deren Behandlung beschrieben.

Nicht erfüllbare Requirements
Obwohl die Menge an Requirements bekannt ist, kann der Fall eintreten, dass das Repository keine Node Types mit passender Capability enthält. Können diese Requirements nicht durch die TOSCA-Laufzeitumgebung verarbeitet werden, bleibt nur die Möglichkeit den Nutzer auf diese Situation hinzuweisen, sodass er die notwendigen Node Types im Repository anlegen kann.

Nicht provisionierbare Node Templates
Enthält ein Node Template keine Requirements, wird eine Untersuchung auf Provisionierbarkeit durchgeführt. Kommt diese Untersuchung zu dem Ergebnis, dass das Node Template nicht provisionierbar ist, werden mögliche Container-Node Templates gesucht, auf denen es provisioniert werden kann. Findet sich hier kein Container-Node Template kann die Vervollständigung nicht fortgesetzt werden. Auch hier ist also eine Interaktion des Nutzers notwendig, der das Node Template gegebenenfalls austauschen muss.

Anwender-Fehler bei der Modellierung
Bei der Modellierung unvollständiger Topologien können Fehler produziert werden. Beispielsweise kann ein Relationship Template des Typs „Deferred“ mit einem Platzhalter verbunden werden. Derartige Fehlerfälle wurden bei der Implementierung nicht beachtet und können zu unerwünschten Ergebnissen führen. Es wird jederzeit davon ausgegangen, dass eine nach 4.3 gültige Topologie modelliert wurde.

6. Verwandte Arbeiten

In diesem Kapitel werden zu dieser Diplomarbeit verwandte Arbeiten beschrieben. Diese behandeln entweder ähnliche Themenbereiche oder dienen als Grundlage für die Ergebnisse dieser Arbeit.

Der Artikel „Managing the Configuration Complexity of Distributed Applications in Internet Data Centers“ [18] beschäftigt sich ähnlich zu vorliegender Arbeit mit Topologie-Transformationen sowie modellgetriebener Provisionierung und Management von Anwendungen. Dabei werden verschiedene Abstraktions-Ebenen einer Topologie festgelegt, die durch Transformationen ineinander überführt werden. Das Ziel ist es, durch diese Transformationen eine Topologie einer Anwendung mitsamt Infrastruktur und Plattformen zu schaffen, die durch eine „Provisioning Engine“ provisioniert werden kann. Dies entspricht der in vorliegender Arbeit entstandenen Lösung. Bei diesen Transformationen wurde ein Verbindungstyp „Deferred“, analog zu 4.3.5 eingeführt, der während der Transformation durch verschiedene Komponenten ersetzt wird. Dabei wird eine Tiefensuche durchgeführt, um die Topologien zu vervollständigen. Der Ansatz des Deferred-Relationship Templates sowie der Tiefensuche wurden in dieser Diplomarbeit - übertragen auf TOSCA-basierte Topologien - ähnlich gelöst.

In Arnold et al. [13] wird ein patternbasiertes Vorgehen beim Deployment von serviceorientierten Architekturen eingeführt. Dabei wird eine Vorgehensweise für die Beschreibung einer Hosting-Infrastruktur eines Deployments vorgestellt. Um dies umzusetzen werden Pattern formalisiert, die über verschiedene Fähigkeiten (Hohe Verfügbarkeit, Skalierbarkeit, Sicherheit etc.) verfügen. Diese können für ein Deployment verwendet werden. Um diese Pattern darzustellen werden Topologien eingeführt, die ähnlich zu dem in vorliegender Arbeit genutzten TOSCA-Standard als Graphen definiert werden. Dabei soll ein Deployment für ungeübte Nutzer vereinfacht werden. Die hierbei eingeführten Topologien enthalten ebenfalls Requirements und Capabilities, die bestimmen welche Komponenten miteinander verbunden werden können. Eine weitere Ähnlichkeit sind die in Arnold et al. [13] als „VirtualizationUnit“ beschriebenen Topologie-Komponenten. Diese fungieren als Platzhalter für konkret einzufügende Komponenten („Concrete Unit“). Das Matching passender Komponenten geschieht dabei ebenfalls durch Requirements. In dieser Diplomarbeit wurde darüber hinaus ein Ansatz geschaffen, Platzhalter ohne Requirements ersetzen zu können.

Binz et al. [9] behandeln die in 2.2 beschriebenen Enterprise Topologie Graphen. Dabei werden Topologien entworfen, die von der Darstellung sehr große Ähnlichkeit mit dem TOSCA-Standard aufweisen. Diese dienen der Zustandsbeschreibung einer bestehenden Infrastruktur.

Durch die Modellierung strukturierter Topologien und einer anschließenden Provisionierung sollen Infrastrukturen entstehen, die mittels ETGs dargestellt werden können.

Breitenbücher et al. [16] führen die grafische Notation Vino4TOSCA für den sonst nur als XML beschriebenen TOSCA-Standard ein. Diese Notation wird in vorliegender Arbeit zum einen dazu verwendet die Komplexität der Grundlagen- und Konzeptkapitel zu vermindern, zum anderen von Winery genutzt um Topologien grafisch zu modellieren. Daher wurde bei der Implementierung (siehe 5.2) ebenfalls mit Vino4TOSCA gearbeitet.

Eilam et al. [20] stellen ähnlich zu Kalantar et al. [18] ein Vorgehen für das Deployment von zusammengesetzten Applikationen vor, welches als Kombination eines modell- und workflowgetriebenen Ansatzes beschrieben ist. Das Ziel ist es dabei über eine schrittweise Verfeinerung von Topologien eine automatisierte Provisionierung zu ermöglichen. Zu Beginn steht ein Deployment-Modell, welches den gewünschten Endzustand darstellt. Dieses wird in ein Workflow-Modell übersetzt, welches eine Menge an Deployment-Operationen enthält und anschließend in eine Workflow-Engine gegeben werden kann. Bei der Übersetzung eines Deployment-Modells in ein Workflow-Modell werden dabei Operationen gesucht, die ausgeführt werden müssen, um den gewünschten Endzustand zu erreichen.

Nachdem sich dieser Abschnitt mit verwandten Arbeiten bei der Modellierung bzw. Vervollständigung beschäftigte, wird im nächsten Abschnitt auf Arbeiten eingegangen, die eine Provisionierung derartiger Topologien behandeln.

6.1. Provisionierung von TOSCA-Topologien

Eine Provisionierung von Anwendungen kann entweder manuell, semi-automatisiert oder vollautomatisiert geschehen. Bei der manuellen Vorgehensweise müssen die Aufgaben, die für eine Provisionierung notwendig sind, von Hand erledigt werden. Diese umfassen zum Beispiel die Nutzung grafischer Oberflächen von Management Tools, die manuelle Installation von Software auf Betriebssystemen und das Anlegen von Konfigurationsdateien. Bei dieser Vorgehensweise spielen Topologien nur selten eine Rolle. Diese werden wenn überhaupt nur zur Dokumentation angefertigt. Die in vorliegender Arbeit vorgestellten Konzepte dienen deshalb bei einer manuellen Vorgehensweise nur für mögliche Dokumentationsaufgaben.

Geschieht eine Provisionierung semi-automatisiert, werden häufig Skripte eingesetzt, die auf einem technischen Level Management- oder Provisionierungsaufgaben ausführen. Der nicht-automatische Teil umfasst dabei das manuelle Aufrufen dieser Skripte mit bestimmten Parameter. Um einen vollautomatischen Ansatz zu ermöglichen können neuartige Technologien wie Chef[1], Juju[2] oder Puppet[3] eingesetzt werden, die derartige Aufrufe automatisieren. Dabei können Topologien verwendet werden, um die Anwendung zu beschreiben und Skripte an Operationen zu koppeln. Jedoch werden derartige Topologien als Text und nicht in Form eines Graphen dargestellt. Daher können diese Tools nicht direkt von den entwickelten Konzepten dieser Diplomarbeit profitieren. Wettinger et al. [21] stellen jedoch ein Konzept vor, welches den TOSCA-Standard für diese Topologien verwendet und sie auf die skript-fokussierten Tools abbildet. Dadurch wird die Möglichkeit geschaffen, die vorgestellten Konzepte zur Vervollständigung auch mit Technologien wie Chef, Juju oder Puppet zu nutzen, um eine vollautomatisierte Provisionierung auf Basis dieser Technologien zu ermöglichen.

Eine vollautomatisierte TOSCA-basierte Provisionierung wird durch die Plan-Generator-Erweiterung [22] von OpenTOSCA [7] ermöglicht. Diese Erweiterung benötigt dabei vollständige Topologien als Eingabe, um einen TOSCA Build Plan zu generieren, der eine Anwendung vollautomatisiert provisionieren kann. OpenTOSCA ist mit dieser Erweiterung somit ein direkter Abnehmer der im Rahmen dieser Lösung vervollständigten Topologien. Der OpenTOSCA Plan Generator unterstützt dabei auch TOSCA Policys [15]. Dies macht es möglich, vervollständigte Topologien mit Policys zu annotieren, um nicht-funktionale Anforderungen an die Provisionierung zu definieren. Die verschiedenen Konzepte können somit nahtlos integriert werden. Eine weitere Technologie-unabhängige, vollautomatisierte Provisionierung ermöglichen Breitenbücher et al. [23]. Dabei werden Management Planlets verwendet, um einzelne Komponenten einer Topologie zu managen oder provisionieren. Diese Planlets sind als Workflows implementiert, die vollautomatisch auf Basis von TOSCA-Topologien orchestriert werden können. Um dies zu ermöglichen werden vollständige, provisionierbare Topologien benötigt. Die auf Basis von Planlets implementierte „Provi-

[1]http://www.opscode.com/chef/
[2]https://juju.ubuntu.com/
[3]http://puppetlabs.com/

sioning Engine“ ist somit direkter Abnehmer der vervollständigten TOSCA-Topologien. Planlets unterstützen zusätzlich TOSCA Policys [24], welche an vervollständigte Topologien angehängt werden können, um somit auch nicht-funktionale Anforderungen bei der Provisionierung mit einzubeziehen. Muss die Provisionierung vervollständigter Topologien noch individualisiert werden, bieten Planlets eine Möglichkeit, dies durch die Implementierung eigener Provisionierungsoperationen umzusetzen [25]. Dies passiert nach der Vervollständigung und beeinflusst somit die vorgestellten Konzepte der vorliegenden Arbeit nicht. Eine weitere Topologie-basierte Provisioning Engine ist Cafe [26]. Cafe bietet die Möglichkeit, Topologien zu provisionieren und auf einer existierenden Infrastruktur zu deployen. Allerdings unterstützt Cafe den TOSCA-Standard aktuell nicht.

7. Fazit und Ausblick

7.1. Fazit

In diesem Kapitel wird ein abschließendes Fazit über die verfasste Diplomarbeit gegeben.

Ziel dieser Diplomarbeit war es ein Konzept zu entwickeln und umzusetzen, um unvollständige TOSCA-Topologien zu modellieren und automatisiert zu vervollständigen. Dabei sollten Aufwand und notwendiges Know-how des Modellierers minimiert werden.

Diese Aufgabenstellung konnte durch die im Rahmen dieser Arbeit entstandenen Lösungen erfüllt werden. Um dies zu erreichen wurde zuerst ein Konzept entwickelt, um unvollständige Topologien nach verschiedenen Anwendungsfällen zu modellieren. Dabei kann der Modellierer derartiger Topologien durch das Angeben von Bedingungen und Voraussetzungen Einfluss auf die Vervollständigung nehmen, aber auch vollständig auf einen Einfluss verzichten. Dies erlaubt es einerseits ungeübten Modellierern eine Topologie ohne Know-how vervollständigen zu lassen, andererseits aber auch geübten Modellierern die Vervollständigung nach Belieben zu steuern. Die Durchführung der Vervollständigung wurde mit zwei verschiedenen Ansätzen gelöst. Dabei wurden eine Ein-Schritt-Vervollständigung und eine schrittweise Vervollständigung eingeführt. Die Ein-Schritt-Vervollständigung erfordert kaum Einschreiten des Nutzers während der Vervollständigung, erhöht jedoch den Modellierungs-Aufwand. Hierfür muss eine Topologie wie in Kapitel 4.3 beschrieben modelliert werden. Die Verwendung von Platzhaltern oder Relationship Templates des Typs „Deferred" ist bei dieser Vorgehensweise meist notwendig, da oftmals konkrete Anforderungen an Infrastruktur-Komponenten existieren. Bei der schrittweisen Vervollständigung hingegen ist die Aufwands-Verteilung umgekehrt. Während bei der Modellierung auf Platzhalter und Relationship Templates des Typs „Deferred" verzichtet werden kann, müssen die einzufügenden Templates während der Vervollständigung in jedem Schritt ausgewählt werden. Dabei wird dem Nutzer eine Auswahl aller möglicher einzufügender Templates gegeben. Dies fördert die Nachvollziehbarkeit der Vervollständigung und ermöglicht es dem Nutzer diese zu steuern.

Durch die Implementierung der beschriebenen Algorithmen und der Integration in das Winery-Modellierungstool kann die Funktionalität auch in der Praxis umgesetzt werden. Dies steigert die Nutzbarkeit von Winery und hilft ungeübten Nutzern vollständige Topologien grafisch zu modellieren.

Neben den implementierten Konzepten wurden Ideen entwickelt, die aufgrund des großen Aufwands nicht umgesetzt werden können, jedoch für zukünftige Arbeiten und Weiterentwicklungen denkbar sind. Diese werden im folgenden Kapitel „Ausblick“ beschrieben.

7.2. Ausblick

Diese Kapitel beschreibt zwei Ansätze, die im Rahmen dieser Arbeit nicht umgesetzt wurden konnten. Darunter die Verwendung von TOSCA-Policys bei der Vervollständigung sowie das Erfüllen von Eigenschaften bei der Instanziierung von Node und Relationship Templates.

7.2.1. TOSCA-Vervollständigung mit Policys

Bei der Vervollständigung von TOSCA-Topologien können bisher nur funktionale Anforderungen an die zu ergänzende Infrastruktur definiert werden. Sinnvoll ist es jedoch auch nicht-funktionale Anforderungen festlegen zu können. Diese können beispielsweise Vorgaben für die Verfügbarkeit der Komponenten, die Kosten oder den Datenspeicherort enthalten. Typischerweise werden derartige Anforderungen per Policy definiert. Das Format der Policy ist dabei nicht festgelegt, meist handelt es sich um ein XML-Dokument (vgl. WS-Policy [14]). Im Rahmen des Konzeptes wurde beschlossen, ein eigenes Policy-Format einzuführen, welches speziell für die Vervollständigung von TOSCA-Topologien entworfen wurde. Derartige Policys können in Verbindung mit TOSCA für eine Provisionierung von Anwendungen verwendet werden [24].

Im Rahmen der vorliegenden Arbeit wurde bereits ein Konzept einer solchen Policy entwickelt, welches in Anhang A.3 beschrieben ist.

7.2.2. Eigenschaften von TOSCA-Templates

Aufgrund des Umfangs dieser Diplomarbeit kann neben den Policys das Erfüllen der Eigenschaften (TOSCA: „PropertyDefinitions“) beim Instanziieren von Node und Relationship Templates nicht umgesetzt werden. Dazu wäre es notwendig, alle möglichen Eigenschafts-Felder der Node Templates zu kennen und mit festgelegten oder generierten Werten zu befüllen. Dies ist vor allem bei der Generierung von Passwörtern, Benutzernamen etc. sehr komplex und teilweise nicht möglich, da Informationen des Nutzers hierfür notwendig wären. Für zukünftige Arbeiten wäre die Entwicklung eines Konzeptes, um diese Problematik zu lösen, jedoch sinnvoll. Im Rahmen dieser Arbeit werden lediglich, in den Typen festgelegte, Default-Werte eingefügt.

A. Anhang

A.1. XML-Code der Beispiel TOSCA-Topologie aus Abbildung 2.5

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tosca:Definitions xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12"
  id="74fa45a3-2c41-488d-a69f-0d5340d4dabd"
  targetNamespace="http://www.example.org/opentosca/winery/winery">
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/f0ea23b6-5d27-473f-8f26-6e38d6294c6e.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/9fe18830-bbd8-410b-ab84-f89809cb0d47.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/7e4ca9ed-9b02-4aa1-a143-372ddb02dfff.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/1b66568a-15bf-4e37-ba4c-1f705452b5a7.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/aa91280a-6e72-4ef3-b127-43dd50be7607.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.org/opentosca/winery/org.opentosca.winery.repository"
      location="Definitions/repod/03540447-b56a-4db6-a6eb-9cd30e604978.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:Import
      namespace="http://www.example.com/sample"
      location="Definitions/s/MyDefinitions2.definitions"
      importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
    <tosca:ServiceTemplate id="demo" targetNamespace="http://www.example.org/demo">
        <tosca:TopologyTemplate>
            <tosca:NodeTemplate xmlns:tonty="http://www.example.org/tosca/nodetypes"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="My Amazon Instance" id="n5009" type="tonty:Amazon_EC2">
            </tosca:NodeTemplate>
            <tosca:NodeTemplate xmlns:demo="http://www.example.org/demo"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="n7323" id="n7323" type="demo:Virtual Machine">
            </tosca:NodeTemplate>
```

```
            <tosca:NodeTemplate xmlns:tonty="http://www.example.org/tosca/nodetypes"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="n6549" id="n6549" type="tonty:Ubuntu">
            </tosca:NodeTemplate>
            <tosca:NodeTemplate xmlns:ns0="http://www.example.org/toscaprimer"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="n1299" id="n1299" type="ns0:ApacheWebServer">
            </tosca:NodeTemplate>
            <tosca:NodeTemplate xmlns:tonty="http://www.example.org/tosca/nodetypes"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="n8370" id="n8370" type="tonty:PHP">
            </tosca:NodeTemplate>
            <tosca:NodeTemplate xmlns:tonty="http://www.example.org/tosca/nodetypes"
              xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
              name="n1459" id="n1459" type="tonty:PostgreSQL">
            </tosca:NodeTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_36" type="s:processDeployedOn">
                <tosca:SourceElement ref="n6549"/>
                <tosca:TargetElement ref="n7323"/>
            </tosca:RelationshipTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_41" type="s:processDeployedOn">
                <tosca:SourceElement ref="n7323"/>
                <tosca:TargetElement ref="n5009"/>
            </tosca:RelationshipTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_47" type="s:processDeployedOn">
                <tosca:SourceElement ref="n8370"/>
                <tosca:TargetElement ref="n1299"/>
            </tosca:RelationshipTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_52" type="s:processDeployedOn">
                <tosca:SourceElement ref="n1459"/>
                <tosca:TargetElement ref="n6549"/>
            </tosca:RelationshipTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_56" type="s:processDeployedOn">
                <tosca:SourceElement ref="n1299"/>
                <tosca:TargetElement ref="n6549"/>
            </tosca:RelationshipTemplate>
            <tosca:RelationshipTemplate xmlns:s="http://www.example.com/sample"
              id="con_60" type="s:processDeployedOn">
                <tosca:SourceElement ref="n8370"/>
                <tosca:TargetElement ref="n1459"/>
            </tosca:RelationshipTemplate>
        </tosca:TopologyTemplate>
    </tosca:ServiceTemplate>
</tosca:Definitions>
```

A.2. Typ-Umfang der Testfälle

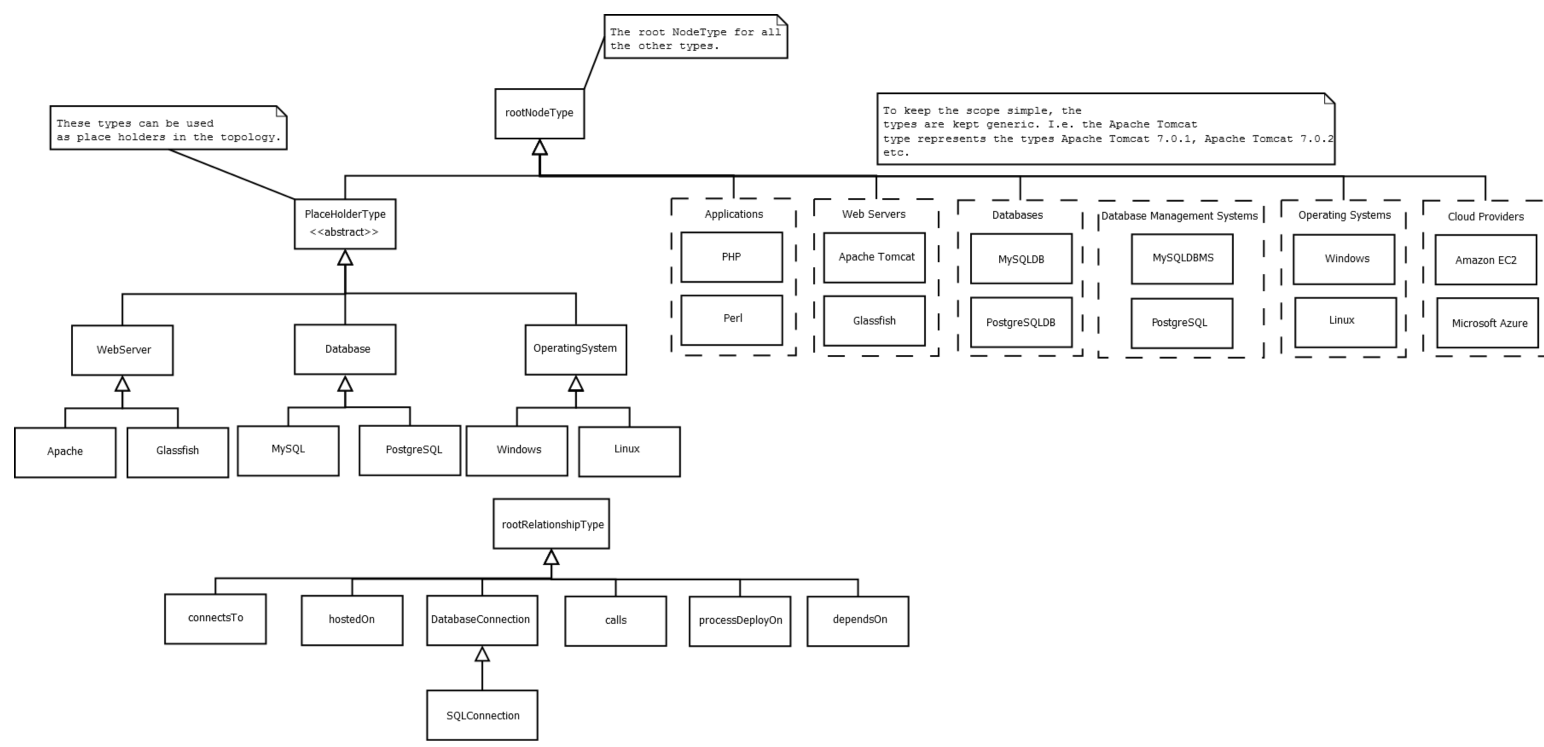

Abbildung A.1.: Typ-Umfang Node und Relationship Types

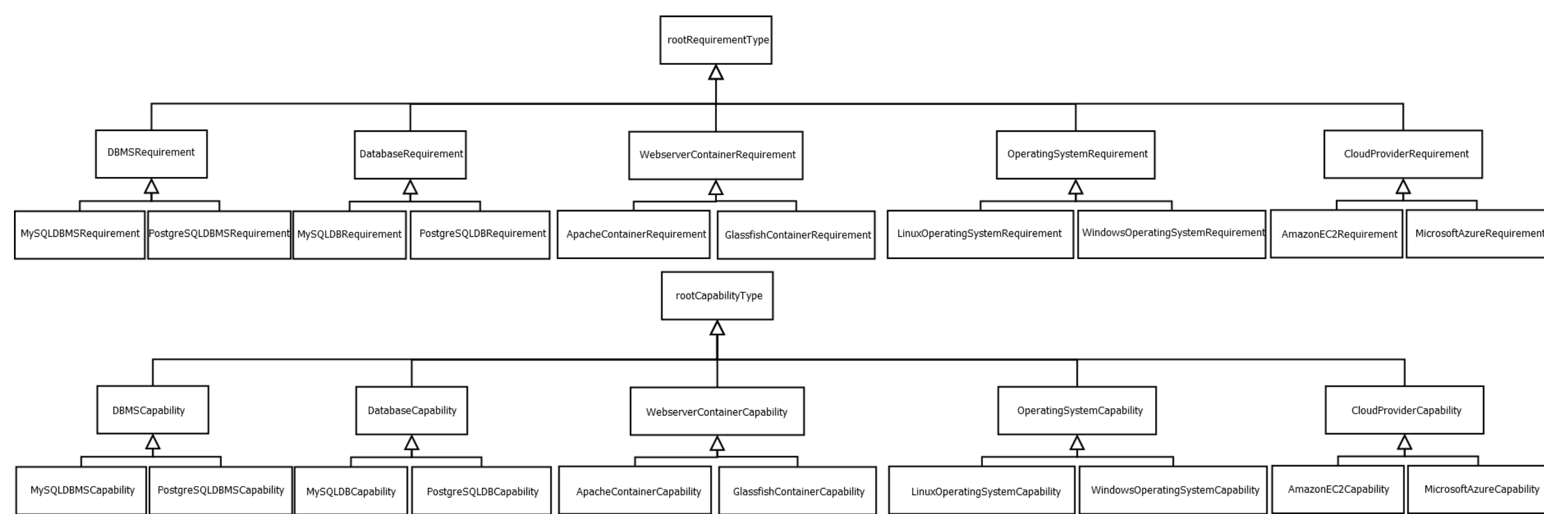

Abbildung A.2.: Typ-Umfang Requirement und Capability Types

A.3. TOSCA-Policy Format

Für die Vervollständigung von Topologien können neben der modellierten, unvollständigen Topologie nicht-funktionale Anforderungen an die Vervollständigung benötigt werden (bspw. SLA-Inhalte). Diese spalten sich in technische und nicht-technische Anforderungen auf. Im Folgenden werden Beispiele möglicher Anforderungen aufgezählt:

- Technische Anforderungen
  - Cloud-Deployment-Modell
  - Anbieter-Präferenz
  - Betriebssystem-Präferenz
  - Gewünschte Anzahl verwendeter Maschinen
- Nicht-technische Anforderungen
  - Maximale Kosten
  - SLA-Inhalte wie Verfügbarkeit oder Skalierbarkeit

Derartige nicht-funktionale Anforderungen können jedoch nicht in der TOSCA-Topologie modelliert werden. Um diese dennoch im CSAR abspeichern zu können, werden sie mittels TOSCA-Policy (siehe 2.3.5) in das Definitions-Dokument eingefügt. Hierfür wird als erstes ein TOSCA-Policy Type eingeführt, der die Struktur der abgespeicherten Informationen vorgibt. Da bei der Vervollständigung ein eigenes Policy-Format entwickelt wird, muss dieses zunächst per XML-Schema definiert werden. Hierfür bietet TOSCA ein „xs:any“-Element innerhalb des „Types“-Elements im Definitions-Dokument an. An dieser Stelle kann das Schema einer XML-basierten Policy frei definiert werden. Die Definition des Policy-Schemas sollte dabei in einem separaten Definitions-Dokument geschehen, welches falls benötigt über den Import integriert wird. So kann die Wiederverwendbarkeit der Policy gesichert werden. Die in vorliegender Arbeit entstandene Policy gliedert sich in technische und nicht-technische Anforderungen, die mittels XML-Elementen definiert werden. Pro Anforderung wird genau ein XML-Element angelegt. Dies vereinfacht die Lesbarkeit und die Komplexität der entstandenen Policy und fördert die Erweiterbarkeit. Die Definition des TOSCA-Policy Types „Completion Policy“ wird im Folgenden als XML dargestellt:

```
<xml>
  <tosca:Definitions id="PolicyDefinitions" name="Completion Policy">
    <tosca:Types>
      <xs:schema>
        <xs:element name="CompletionProperties">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="DeploymentModel" type="xs:string"/>
              <xs:element name="CloudProvider" type="xs:string" minOccurs="0"/>
              <xs:element name="OperatingSystem" type="xs:string" minOccurs="0"/>
              <xs:element name="NumberOfMachines" type="xs:integer" minOccurs="0"/>
              <xs:element name="SLAContent" type="tSLAContent"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:complexType name="tSLAContent">
          <xs:sequence>
            <xs:element name="MaximalCosts" type="xs:string"/>
            <xs:element name="Availability" type="tPercentage"/>
          </xs:sequence>
        </xs:complexType>
        <xs:simpleType name="tPercentage">
          <xs:restriction base="xs:integer">
            <xs:minInclusive value="0"/> <xs:maxInclusive value="100"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:schema>
    </tosca:Types>
    <tosca:PolicyType name="CompletionPolicyType">
      <PropertiesDefinition element="ns:CompletionProperties"/>
    </tosca:PolicyType>
  </tosca:Definitions>
</xml>
```

Listing A.2: „CompletionPolicy“ als XML

Aus diesem Policy Type können konkrete Policy Templates abgeleitet werden, mit denen die abgefragten Informationen in eine Topologie eingefügt werden können. Das instanziierte Policy Template muss dabei dem, im Typ definierten, Schema entsprechen und alle festgelegten Pflichtfelder erfüllen. Ein solches Beispiel-Policy-Template wird im Folgenden dargestellt.

```
1  <xml>
2    <tosca:Definitions id="PolicyTemplate" name="PolicyTemplate">
3      <PolicyTemplate id="CompletionPolicy" name="Completion Policy" type="bpt:CompletionPolicyType">
4        <Properties>
5          <CompletionProperties>
6            <DeploymentModel> Public </DeploymentModel>
7            <CloudProvider> Amazon EC2 </CloudProvider>
8            <OperatingSystem> Ubuntu 10 </OperatingSytem>
9            <NumberOfMachines> 1 </NumerOfMachines>
10           <SLAContent>
11             <MaximalCosts> 200 </MaximalCosts>
12             <Availability> 99 </Availability>
13           </SLAContent>
14         </CompletionProperties>
15       </Properties>
16     </PolicyTemplate>
17   </tosca:Definitions>
18 </xml>
```

Listing A.3: Policy-Template für die Vervollständigung

A.3.1. Policys im Vervollständigungs-Algorithmus

Während der Vervollständigung kann diese Policy für die Auswahl von Node Templates genutzt werden. Werden Node Templates über die Provisioning-API abgefragt, wird an dieser Stelle neben der Topologie auch die Policy im XML-Format an die API übergeben. Diese soll daraufhin nur Policy-konforme Node Templates liefern. Bei der Suche nach Node Templates im Repository kann die Policy ebenfalls genutzt werden, um passende Node Templates finden zu können. Gibt die Policy beispielsweise vor, dass eine bestimmte Kostengrenze nicht überschritten werden darf, werden zu teure Cloud-Provider in diesem Schritt aussortiert. Realisierbar ist dies mit einem XML-Parser, welcher die Anforderungen aus der Policy extrahiert und die einzufügenden Node Templates auf Einhaltung dieser überprüft.

Literaturverzeichnis

[1] Organization for the Advancement of Structured Information Standards. Topology and orchestration specification for cloud applications. http://www.tosca-open.org/. (Zitiert auf den Seiten 3, 11 und 15)

[2] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery – a modeling tool for TOSCA-based cloud applications. In *ICSOC*, LNCS. Springer, 2013. (Zitiert auf den Seiten 3, 8, 11, 16, 27, 34 und 84)

[3] Klaus Manhart. Was hätten sie denn gerne: Iaas, paas oder saas? http://blog.qsc.de/2012/12/was-hatten-sie-denn-gerne-iaas-paas-oder-saas/, 2012. (Zitiert auf den Seiten 8 und 14)

[4] Organization for the Advancement of Structured Information Standards. Topology and orchestration specification for cloud applications. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html, 2013. (Zitiert auf den Seiten 8, 16, 17, 18, 22, 23 und 25)

[5] Organization for the Advancement of Structured Information Standards. Tosca primer. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.pdf. (Zitiert auf den Seiten 8, 16, 17 und 37)

[6] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. Portable cloud services using tosca. *IEEE Internet Computing*, 16(03), 2012. (Zitiert auf den Seiten 11 und 16)

[7] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In *ICSOC*, LNCS. Springer, 2013. (Zitiert auf den Seiten 11, 16, 27, 74 und 101)

[8] Elisabeth Keller-Stoltenhoff. Das servicelevelagreement (sla) aus rechtlicher sicht - vertragliche regelung wiederkehrender it-dienstleistungen. http://www.gulp.de/kb/lwo/vertrag/servicelevelagreement.html. (Zitiert auf Seite 13)

[9] Tobias Binz, Christoph Fehling, Frank Leymann, Alexander Nowak, and David Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Proceedings of 2012 IEEE International Conference on Cloud Computing*, pages 1–8. IEEE Computer Society Conference Publishing Services, Juni 2012. (Zitiert auf den Seiten 15 und 99)

[10] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*, pages 0–9. IEEE Computer Society Conference Publishing Services, Dezember 2013. (Zitiert auf Seite 15)

[11] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Advanced Web Services. Springer, New York, January 2014. (Zitiert auf Seite 16)

[12] Organization for the Advancement of Structured Information Standards. https://www.oasis-open.org/. (Zitiert auf Seite 16)

[13] William Arnold, Tamar Eilam, Michael Kalantar, AlexanderV. Konstantinou, and AlexanderA. Totok. Pattern based soa deployment. In BerndJ. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2007. (Zitiert auf den Seiten 16 und 99)

[14] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Zitiert auf den Seiten 16, 17, 26 und 104)

[15] Tim Waizenegger, Matthias Wieland, Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Bernhard Mitschang, Alexander Nowak, and Sebastian Wagner. Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing. In Robert Meersman, Herve Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Dou Deijing, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, volume 8185 of *Lecture Notes in Computer Science (LNCS)*, pages 360–376, Heidelberg, September 2013. Springer Berlin Heidelberg. (Zitiert auf den Seiten 17, 26 und 101)

[16] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and David Schumm. Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *Proceedings of the 20th International Conference on Cooperative Information Systems (CoopIS 2012)*, Lecture Notes in Computer Science. Springer-Verlag, September 2012. (Zitiert auf den Seiten 17, 21, 27 und 100)

[17] Institut für Architektur von Anwendungssystemen. Instituts-webseite. http://www.iaas.uni-stuttgart.de/. (Zitiert auf Seite 27)

[18] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *Communications Magazine, IEEE*, 44(3):166–177, 2006. (Zitiert auf den Seiten 43, 99 und 100)

[19] John Hunt and Chris Loftus. Java server pages. In *Guide to J2EE: Enterprise Java*, Springer Professional Computing, pages 365–375. Springer London, 2003. (Zitiert auf Seite 84)

[20] T. Eilam, M. Elder, A.V. Konstantinou, and E. Snible. Pattern-based composite application deployment. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 217–224, 2011. (Zitiert auf Seite 100)

[21] Johannes Wettinger, Michael Behrendt, Tobias Binz, Uwe Breitenbücher, Gerd Breiter, Frank Leymann, Simon Moser, Isabell Schwertle, and Thomas 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); Aachen, Germany, May 8-10, 2013*, pages 437–446. SciTePress, Mai 2013. (Zitiert auf Seite 101)

[22] Kalman Kepes. Konzept und Implementierung eine Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA. Bachelorarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Juli 2013. (Zitiert auf Seite 101)

[23] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. Pattern-based Runtime Management of Composite Cloud Applications. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*. SciTePress Digital Library, Mai 2013. (Zitiert auf Seite 101)

[24] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and Matthias Wieland. Policy-Aware Provisioning of Cloud Applications. In *SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies*, pages 86–95, Stuttgart, August 2013. IARIA. (Zitiert auf den Seiten 102 und 104)

[25] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*, volume 8185 of *Lecture Notes in Computer Science*, pages 130–148, Stuttgart, September 2013. Springer Berlin Heidelberg. (Zitiert auf Seite 102)

[26] Ralph Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, August 2010. (Zitiert auf Seite 102)

Alle URLs wurden zuletzt am 13. Dezember 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift