Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Heng Cao

Eine Studie über Verfolgbarkeit und Interoperabilität mit OSLC anhand von Papyrus

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heng Cao

Eine Studie über Verfolgbarkeit und Interoperabilität mit OSLC anhand von Papyrus

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer : Frau Prof. Zhen Ru Dai
Zweitgutachter : Frau Prof. Birgit Wendholt

Abgegeben am 16.09.2013

Heng Cao

Thema der Bachelorarbeit

Eine Studie über Verfolgbarkeit und Interoperabilität mit OSLC anhand von Papyrus

Stichworte

Interoperabilität, Verfolgbarkeit, OSLC, Papyrus

Kurzzusammenfassung

Interoperabilität und Verfolgbarkeit spielen eine sehr wichtige Rolle in der Softwareentwicklung. Allerdings ist der Aufwand für dessen Erstellung bis jetzt sehr hoch. Diese Arbeit stellt einen Prototype vor, der auf der Basis von Open Service for Lifecycle Collaboration (OSLC) erstellt wird, um die Verfolgbarkeit und Interoperabilität zu demostrieren.

Heng Cao

Title of the paper

A study of traceability and interoperability with OSLC based on papyrus

Keywords

Interoperability, Traceability, OSLC ,Papyrus

Abstract

Interoperability and traceability play a very important role in software development. However, the effort for its creation is very high up to now. This paper presents a prototype based on open service for Lifecycle Collaboration (OSLC) to demonstrate the traceability and interoperability

Inhaltsverzeichnis

1 Einleitung 6

1.1 Problemstellung 6

1.2 Lösungsansatz 7

2 Technologie in OSLC 9

2.1 Linked Data 9

2.1.1 Die Gründe für Linked Data 9

2.1.2 Prinzipien von Linked Data 11

2.2 RDF 12

3 Verfolgbarkeit, bessere Interoperabilität 15

4 Verwandte Arbeiten 16

5 Der Prototype 19

5.1 Ausgangsszenario 19

5.2 Systemabgrenzung 20

5.3 Spezifikation 22

5.4 Entwurf 22

5.4.1 Abstrakt 22

5.4.2 Eclipse Projekte 23

5.4.3 PapyrusCommon 25
5.4.3.1. Komponete 25
5.4.3.2. Komponente Papyrus Resource 25
5.4.3.3. Komponente OSLC Resources 27
5.4.4 Papyrus Provider 30
5.4.4.1. Komponente 30
5.4.4.2. Komponente Resource Access 30
5.4.4.3. Komponete Persistence 33
5.4.4.4. Komponente OSLC Resource Serivce 33
5.4.4.5. Komponete Servlet 34
5.4.5 Papyrus Consumer 35
5.4.5.1. Komponenten 35
5.5 Implementation 37
5.5.1 Konfiguration 37
5.5.2 Probleme 37
5.5.3 PapyrusCommon Implementation 38
5.5.3.1. Klassen , Methode 39
5.5.3.2. Annotation 39
5.5.4 Papyrus Provider Implemenation 41
5.5.4.1. Komponete Persistence 43
5.5.4.2. Komponete ResourceAccess 44
5.5.5 PapyrusConsumer Implementation 47
5.5.5.1. Klassen Und Methode 47
5.6 Test 49
6 Zusammenfassung 53
7 Quellenangabe 55

1 Einleitung

1.1 Problemstellung

In modernen Software Projekten sind die genutzten Tools kaum noch voneinander isoliert. Dies bedeutet, dass während der Softwareentwicklung sehr häufig Daten zwischen Tools ausgetauscht werden müssen, oder eine Abhängigkeit zwischen ihnen gebildet werden müssen. Aus diesem Grund ist die Interoperabilität der Tools immer wichtiger geworden.

Es ist relativ einfach ein gutes Tool zu bauen, aber die Interoperabilität der Tools ist viel schwieriger zu realisieren. Bis jetzt hat man verschiedene Lösungsansätze versucht, um den Datenaustausch und die Interoperabilität von Tools zu ermöglichen. Wie zum Beispiel “single repository”, hier benutzt man einen Repository für zentrale Datenspeicherung, und alle Tools greifen auf das Repository zu, um die Daten von anderen Tools abzurufen. Das ergibt eine einfache und bequeme Lösung wie die unterschiedlichen Tools miteinander kooperieren. Aber dies hat auch Grenzen. Die Tools arbeiten an Daten, welche eine Kopie der Originaldaten aus dem Repository sind. Diese Daten sind wiederum lokal gespeichert. Dies führt dann dazu, dass während der Bearteitung dieser Daten man nicht sicher sein kann, dass die Daten noch aktuell sind. Darüberhinaus kann es leicht zu Konflikten führen, wenn man an einer Kopie arbeitet, falls eine pararelle Verarbeitung dieser Daten existiert. Zusätzlich besteht das Risiko, dass wenn das Repository ausfällt, es eine Verzögerung der Entwicklung gibt, bis es wieder hergestellt ist, da währenddessen keine aktuellen Daten abgerufen werden können. Natürlich gibt es Maßnahmen dagegen, wie Spiegelung des Repository als Backup, aber dies bringt wieder das Problem mit Aktuallität mit sich, sowie den extra Aufwand für die Wartung des Backups. Je größer das Projekt ist, desto anfälliger ist die Lösung von “single repository”.
Eine häufig eingesetzte dezentrale Lösung ist “Point to Point”. Anhand von Adaptern kann jedes Tool direkt ein anderes Tool ansprechen. Allerdings ist dies besser dafür geeignet, wenn die Anzahl der beteiligeten Tools gering ist, und kaum noch weitere Tools hinzufügt werden müssen, da die Komlexitität der Integrataiton hier N^2 ist.

Da der benötigte Adapter für Tools expontiell wächst, ist ein weiteres Problem dabei auch, dass falls sich ein Tool verändert, dann müssen die Adapter von allen anderen Tools angepasst werden. Damit wächst der Arbeitsaufwand extrem.

In dieser Arbeit handelt es sich um die Zusammenarbeit von: generic Process Management (gPM), Dynamic Object Oriented Requirments System (DOORS) und Papyrus, sowie um die Interoperabilität von Daten, welche von den drei Tools in Anspruch genommen wird. Als Beispiel: wenn in gPM ein Prozess verändert oder hinzufügt wird, sollen dann die verwandten Daten z.B ein Zeichnung in Papyrus und Anforderungen in REq schnell gefunden und eventuell angepasst werden. Obwohl alle drei Tools von dem selben Hersteller stammen, automatisch können sie die Daten der anderen nicht verwenden. In dieser Arbeit wird daher hierfür einen Lösungsansatz präsentiert.

1.2 Lösungsansatz

Wie im vorherigen dargestellt, waren die bisherigen Lösungsansätze nicht zufriedenstellend. Daher wurde Im Jahr 2008 Open Service for Lifecycle Collaboration (folgend als OSLC bezeichnet) ins Leben gerufen. Dies ist eine Gemeinschaft von Softwareentwicklern, Operationsexperten und Organisationen wie IBM, welche anstreben, die Spezifikation für integrierende Tools zu erstellen . Diese sollen erlauben unabhängige Software und Lebenszyklustools in ihre Daten und Workflows sehr leicht zu integrieren.

Dabei hat sich die OSLC Gemeinschaft in mehrere Gruppen aufgeteilt, welche sich mit eigenen Bereichen, wie Change Management, oder Requirement Management beschäftigen. Diese werden in OSLC als Domain bezeichnet. Alle Domains basieren auf dem OSLC Core, so dass die Zusammenarbeit zwischen unterschiedlichen Domains gewährleistet wird.
OSLC basisert außerdem auf W3C Linked Data. Die technische Merkmale davon sind, dass URls als Namen benutzt werden, welches HTTP URls sind, damit man diese Namen nachschlagen kann. Wenn also jemand nach einer bestimmten URI sucht, kann er dirket auch auf nützliche Information, die in Standard Formaten wie RDF gespeichert sind, zugreifen.
In OSLC ist jedes Artefakt im Lifecycle, zum Beispiel eine Anforderung, und dabei gleichzeitig auch eine HTTP Resource, die wiederum durch Standard Methoden von HTTP wie GET oder PUT manipuliert werden kann.

Die Vorteile von OSLC sind dabei, dass für die Kommunikation zwischen den Tools keine speziellen Protokolle erforderlich sind. Durch Representational State Transfer (REST) HTTP können die Daten sehr leicht veröffentlicht und auch darauf zugegriffen werden. Der Aufwand um mehrere Tools zu integrieren ist daher sehr gering, weil alle denselben Standard benutzen und keine weitere Verbindungen benötigt werden, wie bei der Punkt zu Punkt Integration. Da die beteiligende Parteien nicht eingeschränkt sind, bietet OSLC hohe Flexibelität um Softwarelösungen zu entwickeln.

Aus diesen Gründen bietet OSLC für diese Arbeit daher schon ein gutes Fundament. Verfolgbarkeit geht um das Verlinken von verwendeten Resourcen beziehungsweise Artefakten, dafür kann ein OSLC Adapter für Topcased Tools entwicklet werden, zum Beispiel wenn eine neue Anforderung von Topcased Reg erstellt wird. Dabei können die Daten, welche relevant zur Beschreibung und Weiterverarbeitung sind, durch den Adpater einer HTTP Resource kreiert werden. Diese beinhaltet dann den Link zur Originaldatei der neuen Anforderung, sowie der Beschreibungen zu dieser Anforderung, sowie auch die anderen verwandtene Resourcen, welche ebenfalls HTTP-Resourcen sind. Die Verlinkung andere Resourcen kann nachträglich stattfinden.
Um eine detailierte technische Lösungen zu finden, müssen noch folgende Fragen beantwortet werden: wie genau finden die Verlinkungen statt, wann wird die HTTP Resource kreiert, und vor allem wie soll eine Resource aussehen. Desweiteren sollen auch noch die einzelnen Methoden miteinander verlinkt werden können.

2 Technologie in OSLC

OSLC basiert auf mehreren standardisierten Technologien wie REST, Linked Data und RDF. Diese Arbeit enthält keine genaue Erläuterung zu OSLC. Mehr Informationen über die Spezifikation von OSLC findet man auf deren Homepage[12]. Hier werden nur die Technologien, die in OSLC verwendet werden erklärt.

2.1 Linked Data

In diesen Kapitel wird Linked Data vorgestellt, da wie in der Einleitung erwähnt, OSLC auf Linked Data basiert.
Als Linked Data designed wurde, kamen drei Schlüsselfagen auf:
1. Wie am besten die Wiedernutzung von Daten ermöglicht wird;
2. Wie können die in Beziehung stehende Information sich wieder finden;
3. Wie können die Daten von einer Menge von Daten aus einer unbekannten Datenquelle integriert werden?
Wie das World Wide Web unsere Verbundenheit und das Konsumieren von Daten revolutionierte, so kann Linked Data auch revolutionieren wie Daten genutzt, gefunden und integriert werden.

2.1.1 Die Gründe für Linked Data

Um das Konzept von Linked Data zu verstehen, ist es wichtig zu berücksichtigen, wie die Verteilungs- und Wiederbenutzensmechanismen von Daten im Netz sind.
Ein wichtige Faktor von Wiederbenutzung der Daten ist eine gute Struktur. Je mehr regulärer und besser definierte Stukruturen die Daten haben, desto einfacher ist es Tools zu bauen, damit die zuverlässige Verarbeitung der Daten für Wiederbenutzung gewährleistet ist.

Die meisten Webseiten haben mehr oder weniger eine Struktur, ebenso ist die Sprache HTML, in der die Seiten geschrieben werden, eher konzeptiert die textartigen Dokumenten zu strukturieren als die Daten. Wenn die Daten von Text umgeben sind, ist es daher schwierig für Software die Daten aus dem Quellcode der Seite herauszuziehen. Eine sehr häufig benutzte Lösung sind die Web APIs, die den

Zugriff auf struktierte Daten über HTTP Protokoll ermöglicht. Allerdings hat das Aufkommen von Web APIs zu einer Explosion von kleinen, spezialisierten Anwendung geführt, die Daten von verschiedene Quelle nutzen, die wiederum von unterschiedlichen spezifischen API aufgegriffen werden. Während die Vorteile des programmatischen Zugriffs auf strukturierte Daten unbestritten sind, erzeugt die Existenz einer spezialisierten API für jeden Datensatz eine Landschaft, in der erhebliche Anstrengungen erforderlich sind, um jeweils neue Daten in eine Anwendung zu integrieren. Darüber hinraus referenzieren viele Web APIs die Daten nur mit lokale Indentifikation. Wie zum Beispiel eine Produktbezeichnung 43132 außerhalb dem Kontext der Spezifische API sinnlos wäre. Es gibt in diesem Falle keinen Standardmechinismus für das Referenzieren von Daten, die von einer API beschrieben und von einer anderen zurückgeben werden. Daraus folgt dann, dass die von API zurückgegebene Daten als isolierte Fragmente stehen und es fehlen die weiterleitende Links zu den verwandten Daten. Aus diesem Grund, obwohl Web APIs die Daten über das Internet zugänglich machen, veröffentlichen sie die Daten nicht wirklich im Internet, da die Daten nicht wirklich verknüpfbar und auffindbar sind.

Um das Verlinken von Daten über das Internet zu verteilen erfordert es einen Standard-Mechanismus für die Angabe der Existenz und Bedeutung der Verbindungen zwischen den Elementen, die in den Daten beschrieben werden. Dieser Mechanismus wird ermöglicht durch das Resource Descirption Framework (RDF), welches später noch genauer erläutert wird. Eine wichtige Eigenschaft von RDF ist, dass man sehr flexibel die Dinge in der Welt beschreiben kann, wie Menschen, Orte, oder sogar abstrakte Konzepte, und deren Beziehungen zueinandern. Die Beziehung zwischen Dinge sind im wesentlichen die Links. Ein Beispiel dafür wäre ein Buch, welches durch eine API beschrieben ist und das erhältlich ist in einem Buchladen, der durch eine zweite API beschrieben ist. Der Ort von dem Buchladen kann dann wiederum von einer dritten API beschrieben worden sein. RDF erlaubt es uns die Daten so miteinander zu verlinken, dass man leicht Information wiederfinden und wiederbenutzen kann.
Im Vergleich zu HTML Dokumenten und konventionelle WebAPIs, ergibt dies ein Paar sehr wichtige Eigenschaften von RDF.
RDF verlinkt Sachen, nicht nur Dokumente. Wie in dem Buchverkaufsbeispiel oben, verlinkt RDF nicht einfach die Datenfragmente von allen APIs, sonder die in den Datenfragmenten beschriebene Entities, in diesem Fall das Buch, der Buchladen und die Lage.
RDF Links sind typisiert. HTML Links deuten nur an, dass zwei Dokumente in einer Form miteinandern in Beziehung stehen. In den meisten Fällen muss man aber

selbst herausfinden, was für eine Beziehung dies ist. Im Vergleich dazu zeigen die Daten in RDF die Art der Beziehung zueinander an. Nehmen wir den Buchverkauf als Beispiel, dann wären die Beziehungen inetwa wie *meinBuch istErhätlichIn derBuchladen, derBuchladen befindenSichIn meinerStadt.*
Solche Verbindungen zwischen Daten werden nur implizit in XML oder JSON von WebAPIs zurückgegeben. Deshalb ist eine Seite, deren Daten veröffentlicht und verlinkt in RDF sind, wesentlich einfacher für die Datenauffindung.
Wie Hyperlinks im klassischen Web Dokumente zu einem einzigen globalen Informationsraum verbinden, so ermöglicht Linked Data, dass die Links zwischen Elemeten von verschiedenen Datenquellen genau definiert werden können, damit diese Datenquellen zu einem einzigen globalen Datenraum verbunden werden können.
Die Verwendung von Web-Standard und einem gemeinsamen Datemodell ermöglicht die Implemetaion von Anwendungen, die der gesamte Datenraum nutzen kann.

2.1.2 Prinzipien von Linked Data

Die Prinzipien von Linked Data sind, wie schon in Einleitung kurz erwähnt, das URIs als Namen benutzt werden, welche HTTP URIs sind, damit man diese Namen nachschlagen kann. Wenn also jemand nach einer bestimmten URI sucht, kann er dirket auch auf nützliche Information, die in Standard Formaten wie RDF gespeichert sind, zugreifen.

Linked Data befürwortet es grundsätzlich mit URI-Verweisen Sachen zu identifizieren, nicht nur Web-Dokumenten und digitalen Inhalten, sondern auch Objekte der realen Welt und abstrakte Konzepten. Dazu gehören auch greifbare Dinge wie Personen, Orte und Autos, oder auch diejenigen, die mehr abstrakt sind, wie die Beziehungsarten untereinander. Dieses Prinzip kann als Erweiterung des Anwendungsbereichs des Web von Online-Ressourcen, um Objekt oder Konzept der Welt umfassen zu können, angesehen werden.

Das HTTP-Protokoll ist im Web der universellen Zugriff-Mechanismus. Im klassischen Web werden HTTP URIs verwendet um weltweit eindeutige Identifizierung mit einem einfachen, wohlverstandenen Abruf-Mechanismus zu verbinden. Daher verwendet Linked Data grundsätzlich HTTP-URIs, um Objekte und abstrakte Konzepte zu identifizieren, so dass diese URIs über das HTTP-Protokoll zu der Beschreibung der identifizierten Objekt oder Konzept aufgelöst werden können .

Um eine breite Palette von unterschiedlichen Anwendungen und Web-Inhalten verarbeiten zu können, ist es wichtig sich auf standartisierte Content-Formate zu einigen. Die Vereinbarung über die HTML als das dominante Dokumentformat war ein wichtiger Faktor, welche das Web skalierbar macht. Auch Linked Data befürwortet grundsätzlich den Einsatz eines einzigen Datenmodells für die Veröffentlichung von strukturierten Daten im Web - das Resource Description Framework (RDF), ein einfaches Graph-basiertes Datenmodell, das für den Einsatz im Kontext des Web entworfen wurde.

Zuletzt befürwortet Linked Data grundsätzlich noch den Einsatz von Hyperlinks nicht nur zur Verbindung von Web-Dokumenten, sondern auch für jede anderen Art von Dingen. Zum Beispiel kann ein Hyperlink zwischen einer Person und einem Ort, oder zwischen einem Ort und einem Unternehmen eingestellt werden. Im Gegensatz zum klassischen Web, wo Hyperlinks weitgehend untypisiert sind, haben Hyperlinks in einem Linked Data Kontext Typen, welche die Beziehung zwischen den Dingen beschreiben.
Ebenso wie Hyperlinks im klassischen Web Dokumenten in einem einzigen globalen Informationsraum verbinden, so verwendet Linked Data Hyperlinks zu unterschiedlichen Daten, um alles in einem einzigen globalen Daten Raum zu verbinden. Diese Links, ermöglichen wiederum Anwendungen, um die Daten durch den Datenraum zu navigieren.

Zusammengefasst legen die Linked Data Prinzipien die Grundlage für die Erweiterung des Web zu einem globalen Datenraum, welches auf den gleichen architektonischen Prinzipien wie das der klassischen Dokumenten im Web basiert.

2.2 Resource Description Framework (RDF)

RDF ist eine Sprache zum Repräsentieren von Informationen der Resourcen im WorldWideWeb. RDF basiert auf der Idee, dass Daten anhand von Web-Indentifier (URI) identifiziert werden und die Daten durch Ausdrücke mit einfachen Eigentschaften und Werten beschrieben werden.
An diesem Punkt wird die Frage gestellt, warum nicht nur mit XML, was schon in so vielen Umfelden zum Einsatz kommt. Es gibt einen klaren Unterschied zwischen XML und RDF, nämlich die Darstellung von Beziehungen zwischen zwei Resourcen. In XML wird die Beziehung nicht explizit dargestellt. [4]

Abbildung 1 :RDF Beziehung

Was Abbildung 1 darstellt ist ein sehr einfaches Beispiel von zwei Resourcen und deren Beziehung. In XML wird es dargestell wie folgt:

```
<author>
   <uri>page</uri>
   <name>Ora</name>
</author>
```

oder wie

```
<document href="page">
  <author>Ora</author>
</document>
```

oder

```
<document>
  <details>
   <uri>href="page"</uri>
   <author>
      <name>Ora</name>
   </author>
   </details>
</document>
```

Dies sind alles gute XML Dokumente, aber für eine Maschine haben sie alle unterschiedliche Strukturen und werden deshalb auch unterschiedlichen XML Bäume produzieren. Wenn wir ein XML Baum anschauen wie

```
<v>
  <x>
   <y> a="ppppp"</y>
   <z>
      <w>qqqqq</w>
   </z>
  </x>
</v>
```

Dann ist nicht so offensichtlich was die Elemente bedeuten. Vor allem wissen wir nicht wie pppp und qqqq zueinander stehen. Daher ist es viel komplizierter in XML eine Beziehung darzustellen und auszulesen als in RDF.

In RDF/XML, eine XML basierende Syntax in RDF kann es wie folgend aussehen.

```
<?xml version="1.0"?>
 <Page
      xmlns="http://www.w3.org/TR/WD-rdf-syntax#"
      xmlns:s="http://docs.r.us.com/bibliography-info/"
           about="http://www.w3.org/test/page"
           s:Author ="http://www.w3.org/staff/Ora" />
```

Das sind Dinge die mit RDF möglich sind, aber nicht mit XML.
Der semantische Baum kann anaylisiert werden und man erhält am Ende Tripels (die möglicherweise aufeinander verweisen), danach können die Gewünschten benutzt werden unabhängig ob alle anderen von einem selbst verstanden wurden.

Das Problem in XML basiert darauf, dass ein eigenes Verständnis der Struktur benötigt wird, da ansonsten keine semantischen Informationen aus einem Dokument gewonnen werden können, ohne dass das Schema erfasst wurde oder eine Applikation zur Erkennung des Dokumententyps geschrieben wurde.
Wenn ein XML Schema verändert wurde konnten ohne Probleme Zwischenelemente eingefügt werden (z.B. “Details” in den darüberliegenden Baum oder das “div” HTML ist). Dies könnte jedoch jede Abfrage, die auf der Struktur des Dokuments basiert, ungültig machen.
Ohne ein wohldefinertes semantisches Modell kann es zu größeren Problemem kommen.
Diese Probleme können durch ein gutes Design auch in XML ohne RDF vermieden werden. RDF macht diese Dinge allerdings wesentlich einfacher.

3 Verfolgbarkeit, bessere Interoperabilität

Unter Verfolgbarkeit versteht man die Nachvollziehbarkeit von Informationen und die Abhängigkeit zwischen ihnen. Das heißt zum Beispiel, die Art und Weise der Entstehung der Anforderung oder die Umsetzung der Anforderung über ihrem gesamten Lebenzyklus hinweg; die Abhängigkeit, Beziehungen zwischen Anforderungen und andere Artefakte in der Entwicklung.
In der Softwareentwicklung sollte eine größtmögliche Verfolgbarkeit angestrebt werden, da dies enorme Vorteile bringt, zum Beispiel sorgt sie für bessere Dokumentation der Projekte, und kann zeigen ob eine Anforderung erfüllt wird [5].
Grundlegend wird Verfolgbarkeit erreicht durch den Aufbau von Beziehungen zwsichen zwei Artefakten [5].Wenn eine solche Beziehung mit Semantik versehen wird, kann sie dann auch bewertet werden und eine bessere Analyse kann angewendet werden. Dies sorgt für eine bessere Dokumentation, erhöht die Effizienz der Softwareenticklung und erleichtert das Pflegen von Software. Dies wiederum hat eine große Bedeutung in der Softwareentwicklung.
Verfolgbarkeit ist eigentlich kein neues Thema. Viele Organisationen haben in den vergangenen Jahren großen Aufwand betrieben, um ein funktionierendes Verfolgbarkeitssystem aufzubauen. Ziel dabei ist, die Qualität der Softwareentwicklung zu verbessern. Die Projektstakeholder führen kritische Softwareengineering Aktivitäten wie Change impact Analyses ohne Nutzung einer aufgebauten Verfolgung durch. Leider sind viele solcher Systeme unbenutzt. Eine der Hauptursachen hierfür sind fehlende Plattformen und dazugehörige Tools, welche die Verfolgbarkeit beim Einsatz unterstützen.
Ein anderes Problem ist, dass immer größerer Aufwand benötigt wird, um die Beziehung zu erzeugen und zu akualisieren. [5].
Im kommenden Kapitel wird Related Work vorgestellt, sowie einige seiner Einsatzgebiete.

4 Verwandte Arbeiten

Es gibt viele Untersuchungen über Verfolgbarkeit, und viele Vorschläge wie man Verfolgbarkeit realisieren kann. In der Arbeit *Ontology-enhanced description of traceability services* [7]wird ein Ontologie Modell vorgestellt. Dieses Modell erlaubt semantic annotation von Web Service, Ziel ist dabei ein automatischer Webservice für Verfolgbarkeit einer Nahrungsverarbeitungskette.

Das Kern-Ontologie-Modell besteht aus 3 Komponenten, wie in folgender Abbildung zu sehen:

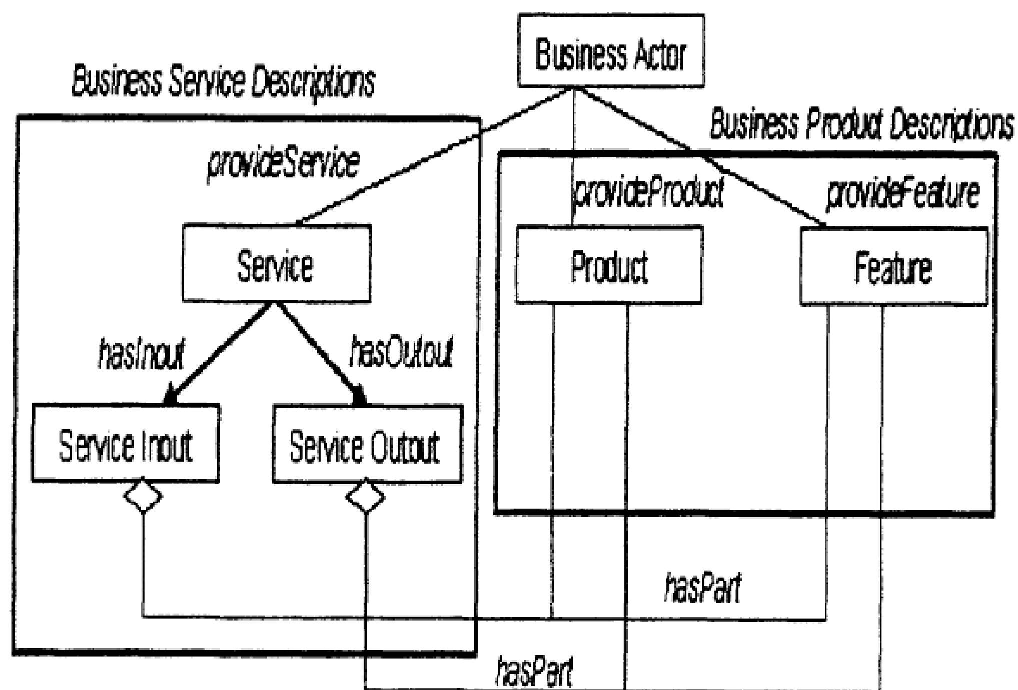

Abbildung 2 : Ontologie-Modell

Der Business Actor ist zum Beispiel ein Produktname oder Produkt Code. Business Service Description beinhaltet das Konzept, welcher Service In-und Output semantisch beschreibt, Business Product Description wiederum beschreibt das Konzept, das den Input und Outservice berechtigt. Die Beziehung zwischen den

Konzepten sind durch verbindende Linien dagestellt, die Pfeilverbindung repräsentieren die funktionalen Beziehungen, die Verbindung ohne Pfeil repräsentiert die one-to-many Beziehung, und die Karoverbindungen stehen für "part-of" Aggregationen.

Die Kern-Ontologie beschreibt das Basiskonzept der Nutzung von Semantic Annotation. Das Kozept kann leicht von ähnlichen Businesskonzept adoptiert werden.
Es gibt allerdings starke Einschränkungen. Diese Ontologie beschreibt nur die Beteiligten in der Verfolgunskette. Dabei orientieren sich dieses Modell an dem vorhandenen Geschäftsmodell, in diesem Fall einer Nahrungsverkaufskette. Außerdem wird in jener Arbeit eigentlich nur ein Modell vorgestellt, nicht die konkrete Umsetzung. Auch wenn man die Traceabilityanwendng nach diesem Ontologie Modell implementiert, muss man immer noch große Änderungen vornehmen, bevor man die Anwendung in einem Geschäftsmodell einsetzen kann. Im Vergleich dazu OSLC, OSLC hat eine hohe Flexibilität. Es bildet nur einen sehr allgemeinen Rahmen, man hat daher große Freiheiten um die Verfolgbarkeit zu ermöglichen, zum Beispiel bleibt die Verfolgung auf der Ojektebene, oder tiefer auf dessen Attributen, je nach Wunsch.

In einer anderen sehr interesanten Arbeit *Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice* [5] wird ein Verfolgbarkeits-informationsmodell vorgeschlagen. In der Praxis werden zur Zeit selten Verfolgbarkeit Meta-Modelle definert und benutzt. Ein Grund hierfür ist das fehlended Wissen über die Vorteile vom Nutzen solcher Modelle, sowie die starke Einschränkung der Anpassungsmöglichkeiten. Jene Arbeit behandelt gezielt diese zwei Probleme. Ziel dabei ist die Verfolgbarkeit wieder auf wesentliches zurückzuführen, sowie eine leichte Praxisumsetzung und -erweiterung zu ermöglichen. Das grundlegend Verfolgbarkeitinformationsmodell beinhaltet zwei Typen von Instanzen, die verfolgbaren Artefakte und die Beziehung zwischen diesen Artefakten. Hier wird dieses Modell in UML dargestellt. Mit Berücksichtigung der Rolle der Artefakte, Abhängigkeit von in Beziehung stehenden Artefakten und Kardinalität; kann die UML Darstellung für das Verfolgbarkeits Modell wie folgend aussehen.

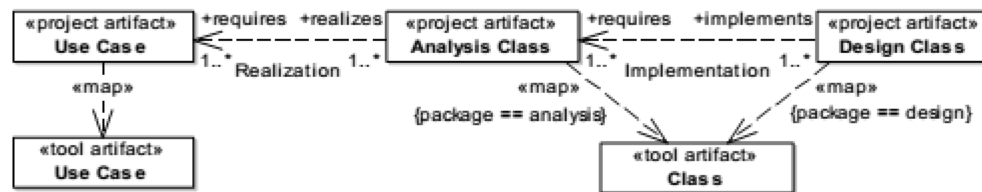

Abbildung 3 : Verfolgbarkeit dargestellt in UML

In jener Arbeit wird nur präsentiert, wie ein einfaches Basis Verfolgbarkeit Informations Modell dargestellt und in die Praxis eingesetzt werden kann. Wenn dies in Entwicklungstools implementiert wird, dann wird es einfacher Verfolgbarkeit zu erzeugen und zu pflegen. Denn erst wenn Verfolgbarkeit aufgebaut wird, kann die Analyses unterstürtzt werden.

Jene Arbeit ist deshalb interessant, weil die Idee, welche in der Arbeit vorgeschlagen wird, wie man die Verfolgbarkeit aufbaut, genau dem Prinzip entspricht, welches OSLC einsetzt. Verfolgbarkeit besteht also aus Artefakten und die Beziehungen zwischen ihnen. Die Vorteile hierbei sind:
Garantierte Konsistenz des Ergebnis in Projekten mit mehreren Stakeholders; Verfolgbarkeit wird auch von Leuten benutzt, die sie nicht erstellt haben. Sie sollen auch in der Lage sein, zu verstehen, wie die Verfolgbarkeit definiert ist, und was man von ihr erwartet; Informationen durch Projekte zu verfolgen ist komplex, und dieses Konzept gibt ein Grundlage an, wie man die Erstellung von Verfolgbarkeit und Validierung von Veränderung vereinfacht.
Auch die zwei Tools IBM DOORS und IBM Rhapsody, die in jener Arbeit erwähnt werden, haben inzwischen OSLC integriert.

5 Der Prototyp

In diesem Kapitel wird die Realisierung des Prototyps erläutert.
Es handlet sich dabei um die Spezifikaiton, Entwurf, Implementation und Test.
Der Prototyp wurde in Eclipse entwickelt.

5.1 Ausgangsszenario

Um die Möglichkeit des Einsatzes von OSLC zu demonstrieren, wird folgendes Szenario, wie in Abbildung 4, zusammen gestellt:

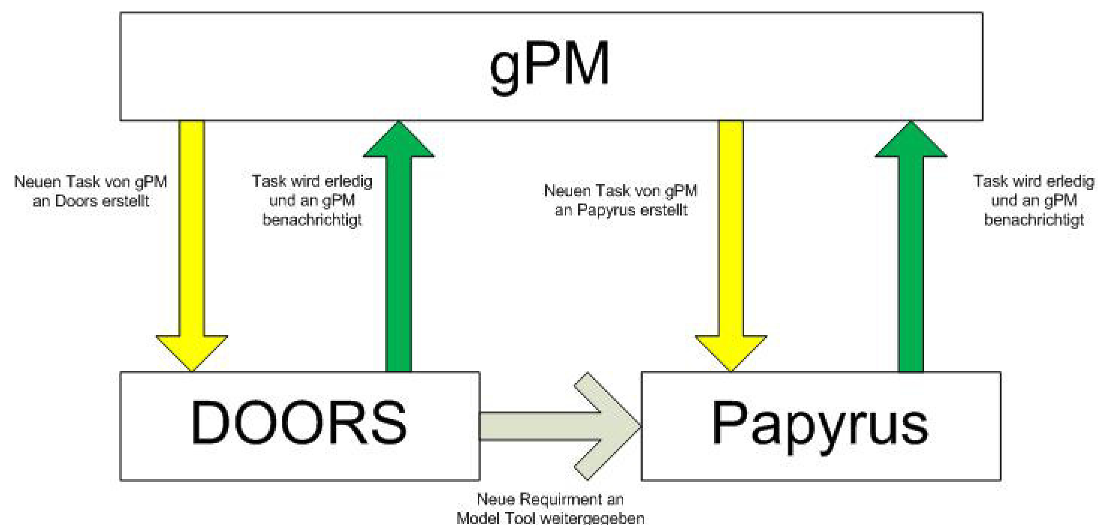

Abbildung 4 : Szenario mit gPM,DOORS und Papyrus

Das Szenario beinhaltet hauptsächlich drei Tools: gPM, DOORS und Papyrus. gPm ist ein Change Management Tool, welches neue Tasks, die neue Change Request repräsentieren erstellt, DOORS ist ein Requirement Mangement Tool und Papyrus ist ein Modellierungstool, welches Uml-Modelle erstellt, zum Beispiel Klassendiagramme. Zuerst werden zwei neue Tasks in gPm erzeugt, welche in der gPM Datenbank gespeichert werden. Die beiden Taske werden dann seperat an DOORS und Papyrurs weitergegeben. DOORS verarbeitet den Task, und erstellt ein neue Anforderung. Nachdem dieser Task erledigt wurde, wird dann gPM benachrichtigt. Die neue Anforderung wird nun auch mit dazugehörigen Modelle in Beziehung gesetzt. Papyrus erhält ebenfalls den Task und führt diesen dann unter

Berücksichtigung der Anforderungen durch. Nach der Erledigung des Tasks wird gPM benachrichtigt. Damit wäre die Kommunikation zwischen den Tools dann erfolgreich abgeschlossen.

Die Kommunikation zwischen den Tools sollen auf Basis von OSLC stattfinden. Das heißt die Tasks werden zuerst zu OSLC Resourcen verarbeitet, und dann greifen DOORS und Papyrus auf sie zu. Genaue so wird es mit den Benachrichtigungen an gMP und den neuen Requirments an Papyrus stattfinden, wie in Abbildung 5 dargestellt.

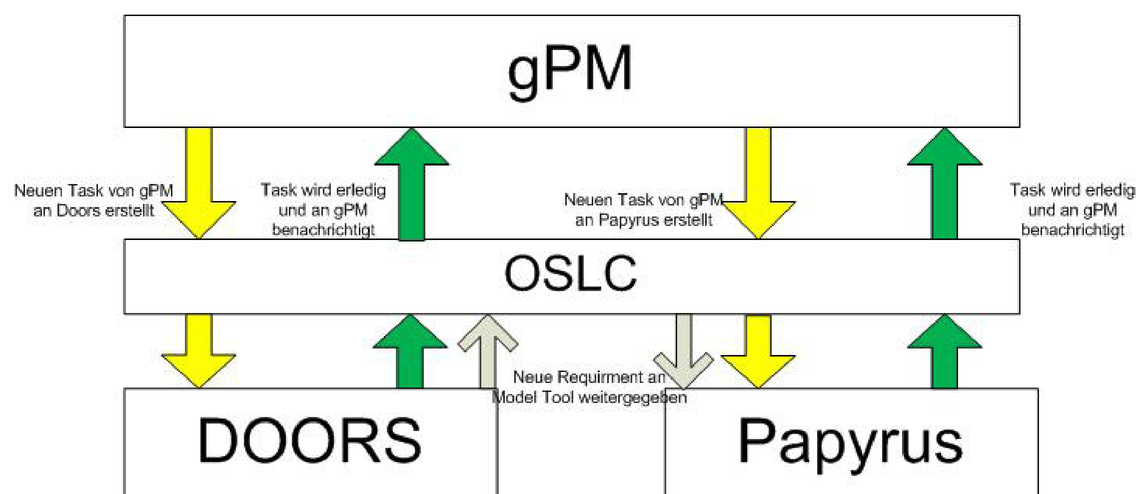

Abbildung 5: Szenario anhand von OSLC

5.2 Systemabgrenzung

Auf Grund fehlender Resourcen und Tools, gPM und DOORS, wird der Prototyp sich in dieser Arbeit hauptsächlich auf den OSLC-Adapter für Papyrus fokusieren. Der Papyrus Service Provider in dem Adpater greift auf eine UML-Datei zu, und veröffentlicht anschließend dessen Inhalt als OSLC-Resouce in RDF/XML Format. Der Papyrus Consumer holt die Resource via HTTP und überträgt die Information in einer andere UML-Datei. Wie in Abbildung 6 und Abildung 7 dargestellt.

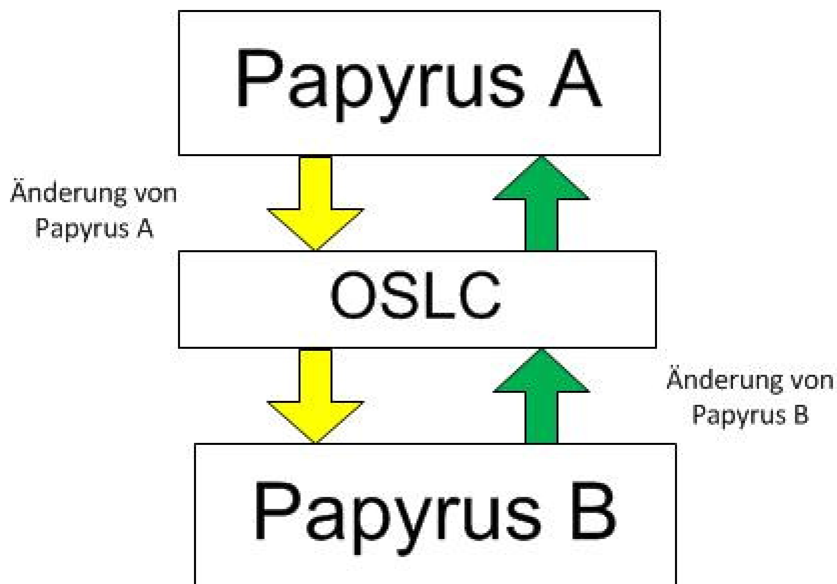

Abbildung 6 : Systemeingrenzung auf Papyrus mit OSLC

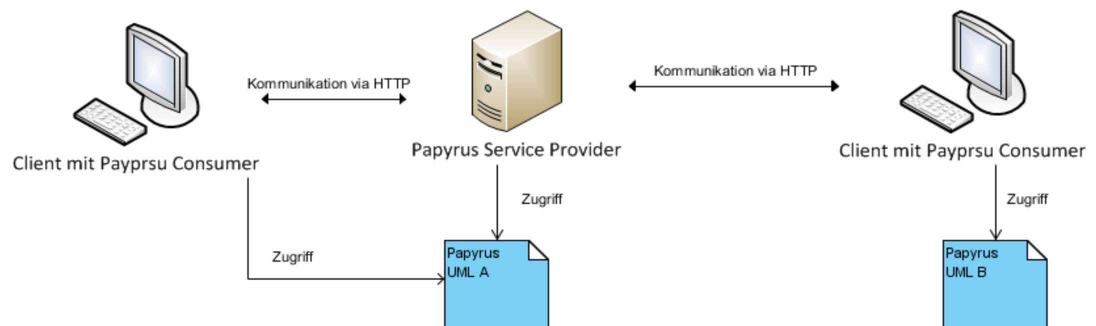

Abbildung 7 : Eingegrenztes Systemaufbau

Um auf die UML Datei von Papyrus zugreifen zu können, wird in diesem Prototyp UML-API benutzt. Grund dafür ist, dass Papyrus-API zurzeit einige Kompabilitätsprobleme hat; mit UML-API kann man auf die UML- Datein zugreifen, ohne weitere Tools öffnen zu müssen.

5.3 Spezifikation

Der Prototyp hat folgende Funktionen: der Papyrus Provider ist in der Lage auf lokale Uml Dateien zuzugreifen, welcher meta-Informationen aus Datein im Klassenobjekt speichert, anschliessend erstellt der OSLC-Provider eine OSLC-Resource und veröffentlicht sie unter einer betimmen URI, welche dann zugreifbar ist per HTTP.
Um die Funktionalität wie Update, Create, Delete von Resource via HTTP zu demonstrieren, wird in einem Client der Papyrus OSLC-Konsument implementiert. Dessen Aufgabe besteht darin, die Information aus derOSLC-Resource zu holen, lokal im Klassenobjekt zu speichern, und ist in der Lage Aktualisierung auf der OSLC-Resouce via Http durchzuführen.
Der Client soll auch die Information aus der OSLC Resource lesen und in UML Datei schreiben können.
Um die Verfolgbarkeit zu simulieren, werden hier sowohl die Papyrus OSLC-Resourcen mit einer URI des anderen Papyrus in Beziehung gesetzt als auch die vorgenommene Änderung dokumentiert.

5.4 Entwurf

5.4.1 Abstrakt

In diesem Abschnitt wird der Entwurf für den OSLC Adapter vorgestellt. Wie in Abbildung 8 dargestellt, auch wenn dies nur eine sehr abstrakt Darstelllung des Adapters und dessen Funktionalität darstellt.

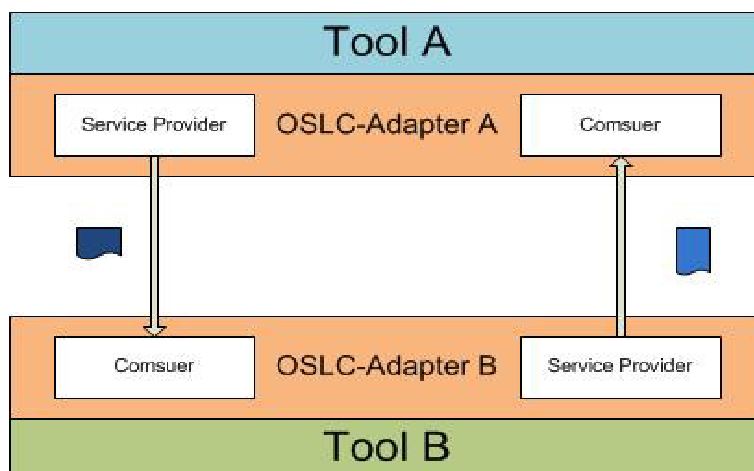

Abbildung 8 : Abstrakter Entwurf

Tool A und B haben jeweils eigene OSLC Adpater. Der Adpdater besteht aus Service Provider und Consumer. Der Service Provider hat die Aufgabe lokalen Resourcen als OSLC-Resource zu veröffentlichen. Der Consumer verabeitet die Resource, welche vom entsprechenden Provider veröffentlichet wird. Daher könnte es mehrere Consumer geben, falls der Adapter mit mehreren anderen OSLC Adapter kommuniziert. Wenn ein Adapter nur Resourcen konsumiert , dann braucht er auch nur den Consumer. Analog dazu, wenn ein Adapter nur Resourcen veröffentlicht wird dann nur der Service Provider benötigt.

Im Service-Provider gibts hauptsächlich zwei Komponenten wie Abbildung 9 darstellt.

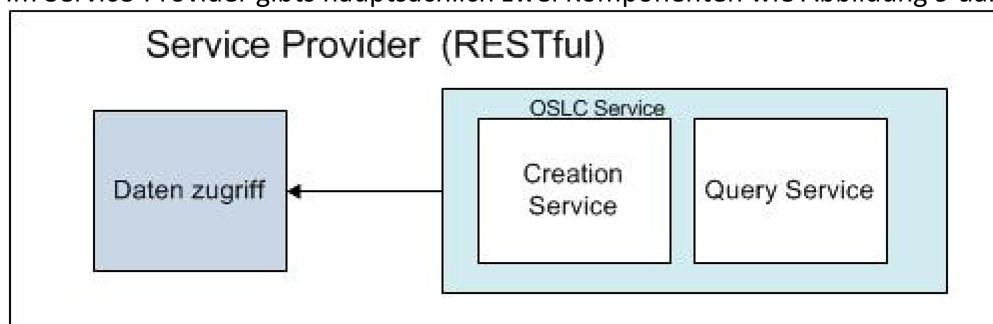

Abbildung 9: Service Provider

Der Datenzugriff sorgt dafür, dass die relevanten Informationenen im Klassenobjekt gespeicht werden und an den OSLC Service weitergeleitet werden. Der OSLC Service hat hier zwei Aufgaben, eine ist anhand von Informationen von Klassenobjekten den OSLC-Service zu erstellen, die andere Aufgabe ist die Verarbeitung von eingehenden HTTP-Abfrage wie GET und POST. Deshalb sollte der Service Provider RESTful sein.

5.4.2 Eclipse Projekte

In Betrachtung von Zuständigkeit und Funktionalität wird der Prototyp in drei Eclipse Projekte geteilt: PapyrusCommon, PapyrusConsumer und PapyrusProvider.

PapyrusCommon

Das Projekt PapyrusCommon bietet fundamentäre Klassen für PapyrusProvider und PapyrusConsumer. Die Objekte der Klassen werden die Information, die von den UML-Datein herkommen erfassen, und stellen diese zur weiteren Verarbeitung zur Verfügung. Ebenso bietet es auch die Klasse, welche für das Erstellen und Erfassen von OSLC Resourcen notwendig ist.

PapyrusConsumer

Der PapyrusConsumer hat die Aufgabe, die Resourcen die vom Papyrus Provider veröffentlicht werden via HTTP-Abfrage zu konsumieren, dabei kann der Consumer nicht nur via REST mit GET die Resource nachfragen, sonder auch mit POST neue Resource erstellen, mit PUT die Resource aktualisieren, und mit DELETE die Resource löschen. Die Informationen von den Resourcen werden in Klassenojekten gespeichert. Solche Trennung hat den Vorteil, dass der PayrusConsumer immer wieder erneut benutzt werden kann.
Im Vergleich zur Entwicklungsmethode, in der man immer neue Apapter entwicklen muss, wenn man neues Tool integrieren möchte, muss man hier für jedes Tool nur einmal entwicklen. Sobald man den Consumer hat, kann man die Information sofort von dem passsenden Provider auslesen.

PapyrusProvider

Der PapyrusProvider ist der Service Provider und sorgt dafür, dass die Metainformation von UML-Datei zuerst gelesen werden. Danach wird die gelesene Information in Klassenobjekten gespeichert. Anschließend wird sie als OSLC-Resource unter einer bestimmten URL zur Verfügung gestellt.

Projektarchitek ist wie Abbildung 10 darstellt.

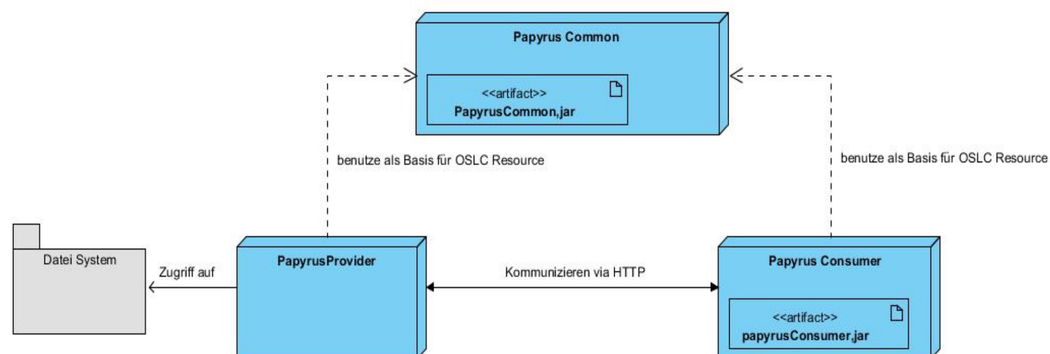

Abbildung 10 : Projektarchitek

5.4.3 PapyrusCommon

5.4.3.1. Komponeten

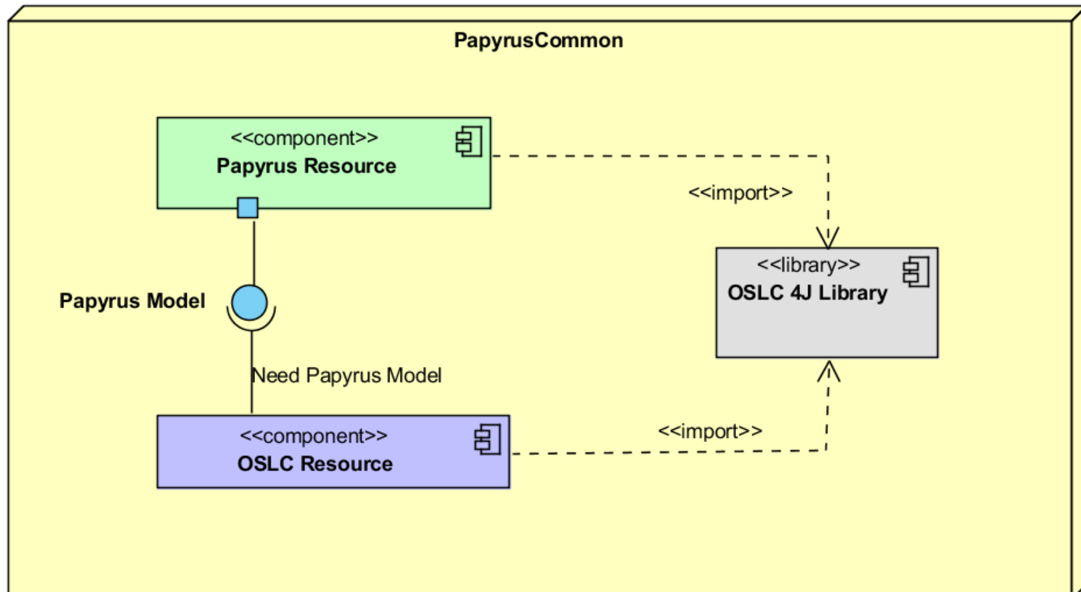

Abbildung 11 : PapyrusCommon Komponenten

Wie in Abbildung 11 dargestellt, hat das *PapyrusCommon* zwei Komponenten.
Beide Komponenten setzen die OSLC 4J Libray vorraus. Die OSLC 4J Library wird hier von Eclipse Lyo [18] breitgestellt.
Die Komponente Papyrus Resource hat die Aufgabe, die Datenstruktur von eingelesenen UML-Dateien darzustellen und die UML Informationen zu speichern. Dabei stellt Papyrus Resource die Klasse *PapyrusModel* für die OSLC Resource zur Verfügung.
Die Komponente OSCL Resource hat die Aufgabe die Papyrus Resource in OSLC Resource umzuwandeln. Dazu kommen auch mehr Informationen über die OSLC-Resource wie related Tasks, die zum Beispiel von gPM erstellt werden. Die Klasse kann nach Bedarf erweitert werden.

5.4.3.2. Komponente Papyrus Resource

Diese Komponente ist der fundamentale Baustein aller andere Projekte. Aufgrund der Systemabgrenzung des Prototypen gibt es insgesamt fünf Klassen. Die fünf Klassen sind in der Lage, das Klassendiagramm zu repräsentieren.

Die Klasse *PapyrusModel* repräsentiert hier das Root Node der Uml Datei. Anhand dieser Klasse ist man in der Lage später auf alle darunter geordneten Diagramme und Beziehungen zuzugreifen.
Das Klassen *Diagramm* stellt in diesem Prototyp eine eingeschränkte Version von einem Klassendiagramm dar. Sie nimmt hier Methode und Attribute der Klassendiagramme auf. Diese Klasse kann als Basisklasse für alle anderen Diagrammtypen angesehen werden. Zum Beispiel kann das Usecasediagramm durch diese Klasse abgeleitet werden.
Für die Attribute und Methoden einer Klasse sind die Klassen *Attribut* und *Operation* zuständig. Hier wird ein Attribut *validatedby* einfügt, um zu zeigen, das eine tiefe Verfolgung auch möglich ist.
Die Klasse *Passciation* ist hier für die Aufnahme von Beziehungen zwischen den beiden Klassendiagrammen in UML zuständig.
Das Klassendiagramm dieser Komponente ist in Abbildung 12 dargestellt.
Die OSLC 4J Library bietet hier die notwendige Annotation, die später sehr wichtig für Erstellung der OSCL-Resource ist. Details werden in dem Kapitel Implementation erläutert.

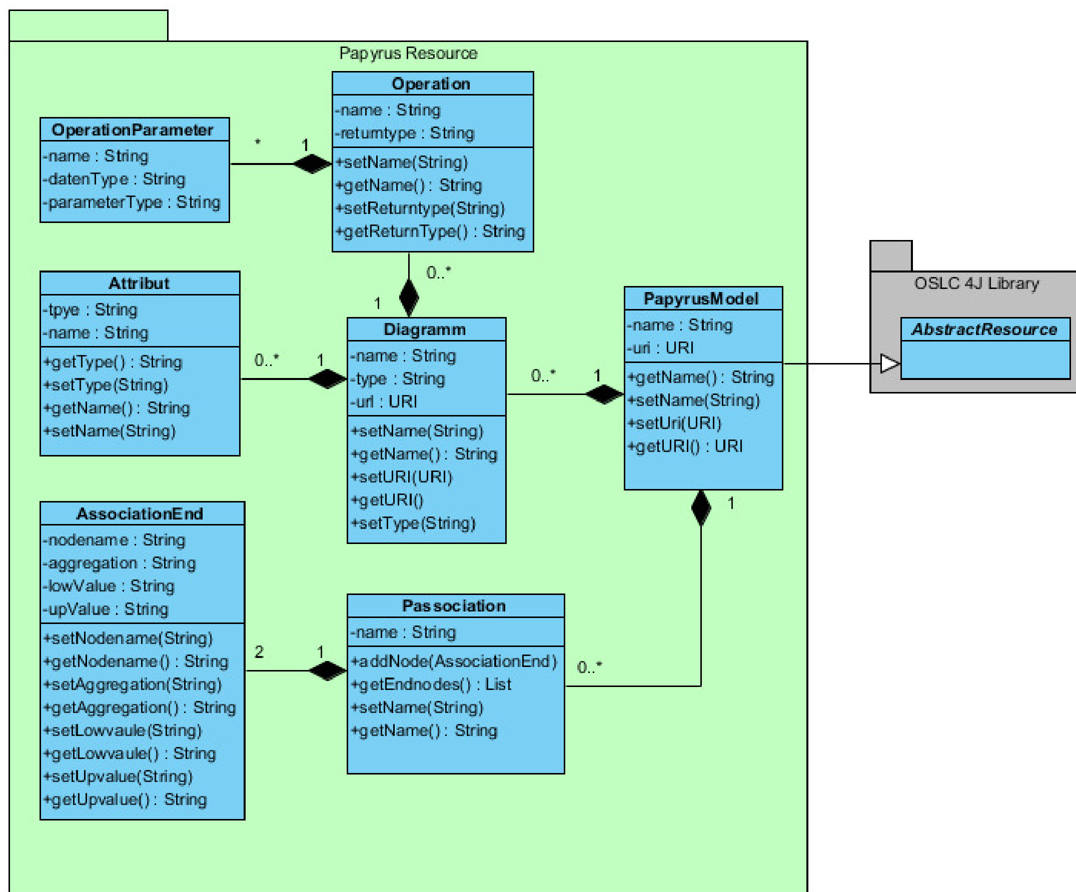

Abbildung 12 : Klassendiagramm von Papyprus Resource

5.4.3.3. Komponente OSLC Resources

In dieser Komponente werden die Information in OSLC-Resourcen repräsentiert.
In der Klasse *Resource* werden zusätzliche Informationen wie Tasks, Beschreibungen und verwandte Artefakte etc. zu den Informationen von Papyrus UML aufgenommen. Die Attribute von diesen Klassen können nach Bedarf beliebig erweitert werden, dies ist dann abhängig davon, was man alles als OSLC-Resource dargestellt haben möchten. Ebenso leitet diese Klasse sich von *AbstractResource* in OSLC 4J Library ab. Das ist die Vorrausetzung dafür, dass eine Klasse als OSLC Resource später im Service Provider veröffentlicht werden kann.

In *PConstants* sind die Konstanten, die für die Definition von Namensraum, Prefix und Annotation notwendig sind.

Die Klasse *Task* kann die Information über Tasks, die zum Beispiel von gPM erstellt werden, aufnehmen.

Die Klassen *RelatedArtefakte* und *ChangeAction* ermöglichen die Verfolgbarkeit in diesem Prototyp. *RelatedArtefakte* ist dafür zuständig, die Beziehung zwischen Artefakte aufzunehmen. Es kann dann zum Beispiel vom Papyrus Modell zurück auf deren Anforderungen oder vorwärs auf die Implementation, oder Tests verfolgt werden. Um eine tiefere Verfolgung zu ermöglichen, das heißt Verfolung zwischen einzelnen Elemente zum Beispiel einer Klasse und einem Testcase innerhalb der OSLC Resourcen, muss das betroffene Element auch in der Lage sein, solche Beziehung aufnehmen zu können. Wie schon die Klasse *Attribute* ist sie ist in der Lage die URI von anderen Artefakte aufzunehmen. Das etabilieren einer Beziehung ist hier also kein Problem. Allerding sollte man hier beachten, dass das Element an sich innerhalb eines OSLC Resource kein eigene URI Verfügen. Aus diesem Grund muss man hier die *ID* von den Elementen und die URI von OSLC zusammen benutzen als URI für ein Element. Es entsteht daher natürlich Aufwand für das Parsen solche URI.
Das Klassendiagramm der OSLC Resourcen ist in Abbildung 13 dargestellt.

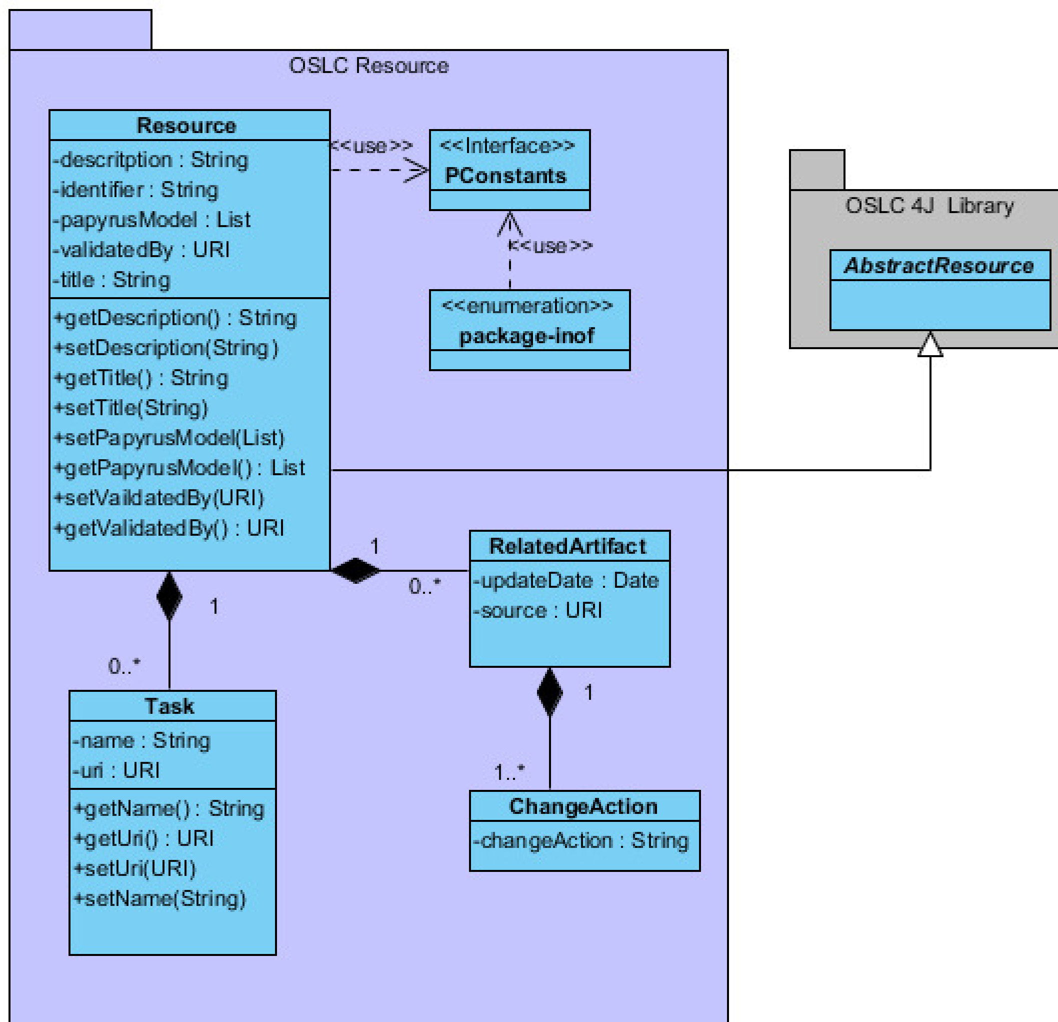

Abbildung 13 : Klassendiagramm von OSLC Resource

Das zusammengesetzte Klassendiagramm der beiden Komponenten wird dargestellt in Abbildung 14. Hier kann man erkennen, dass die Klasse *PapyrusModel* von der Klasse *Resource* benutzt wird.

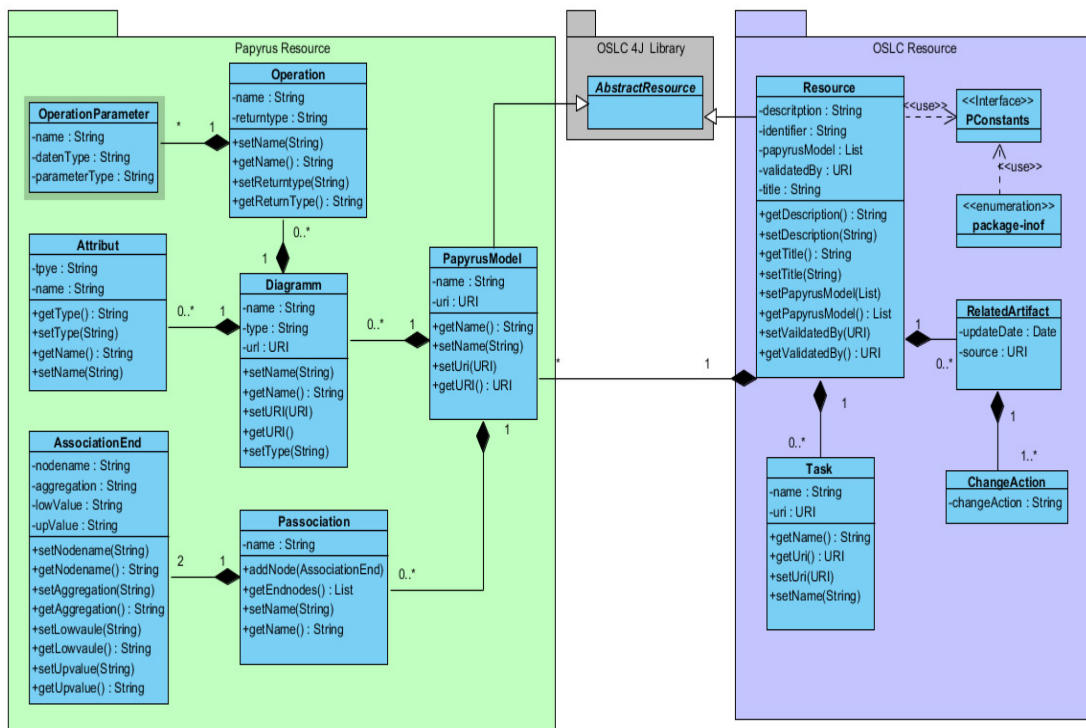

Abbildung 14 : Zusammengesetztes Klassendiagramm von Papyrus Resource und OSLC Resource

5.4.4 Papyrus Provider

5.4.4.1. Komponenten

Der Papyrus Provider ist hier der Papyrus Service Provider. Dessen Aufgabe ist, wie schon im vorherigen Kapitel erwähnt, die lokalen Daten als OSLC Resourcen zu verpacken und anschliessend via HTTP zu veröffentlichen.
Nach der Zuständigkeit wird der Projekt Provider in vier Komponeten aufgeteilt. *Resource Access*, *Persistence*, *OSLC Resource Service* und *Servlet*. Abbildung 15 ist die Komponentenübersicht des Papyrus Providers.

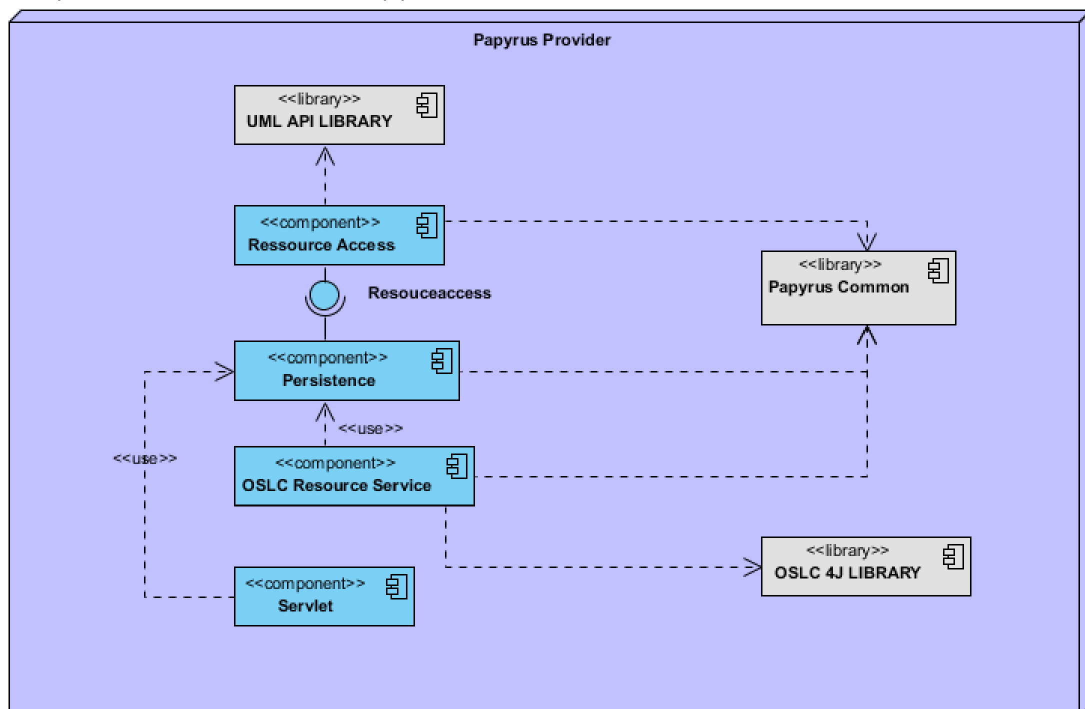

Abbildung 15 : Papyrus Provider Komponenten

5.4.4.2. Komponente Resource Access

Die Komponente *Resource Access* hat die Aufgabe, auf lokale UML-Datei zuzugreifen, und alle benötigten Informationen zu Speichern, und zur weiteren Verwendung bereitzustellen.

Resource Access besitzt eine Klasse, die *ResourceAccess* (siehe Abbildung 16). Diese Klasse bietet alle notwendigen Methoden für das Verpacken von UML Informationen zur OSLC Resource, und die Methode für die Aktualisierung der gelesenen UML Datei. Hier bietet *PapyrusCommon* die wichtige Klassen für das Repräsentieren der

Information einer UML Datei. Die Klassen von *UML API* bietet die Java Schnittstelle für den Zugriff auf eine UML Datei.
Zum besseren Verständnis ist es sinnvoll, dass die Methode in dieser Klasse kurz erläutert werden.
Die Methode *createUMLClass* sorgt dafür, dass eine neue Klasse in UML kreiert wird. Als Parameter braucht diese Methode den Namen der neuer Klasse, das übergeordnete Element, das UML Model, und eine Angabe dazu, ob die Methode abstrakt ist. In diesem Prototyp wird es immer als "false" übergeben, um den Umfang vom Prototpy einzugrenzen. Als Rückgabe wird dann die neue angelegte Class-Instanz zurückgegeben.
Analoge dazu werden die Methode *createUMLClassAttribute*, *createUMLOperation*, *createUMLParameter* ähnliche aufgebaut. Bei *CreateUMLParamter* werden als Übergabeparamter der Name, Datentyp, die übergeordnete Methode, und die Richtung von dem angelegten Paramter angegeben. Wenn der Paramter die Richtung "return" hat, dann entspricht dessen Datentyp dem Rückgabetyp von der Methode.
Die Methode *getResource* sorgt dafür, dass die UML Information von Dateien in den Maps gespeichert wird, als Paramter wird die OSLC Resource übergeben.
Die Methode *initialResource* initiert die OSCL Resource für den Service Provider, wenn der Service Provider startet.
ModelSave ist die Methode, welche die Änderung in der UML Datei speichert.
Bei der Methode *updateOSLCResource* wird die OSLC Resource sowie *getResource* übergeben, der Unterschied ist hier, dass bei *updateOSLCResource* wird zusätzlich noch Information über Herkunft der Änderung der OSLC Resource hinzufügt.
UpdateUML Methode übernimmt die Änderung der OSLC Resoruce, und schreibt sie in die UML Datei hinein.
Mapping zwischen der Information von der UML Datei und der OSLC Resource erfolgt durch Hashmap. Dabei sind die Namen der Schlüssel der Maps, weil die Papyrus UML auf der Datei baisert, aus diesen Grund müssen alle Class-Instanz einen eindeutigen Namen haben.

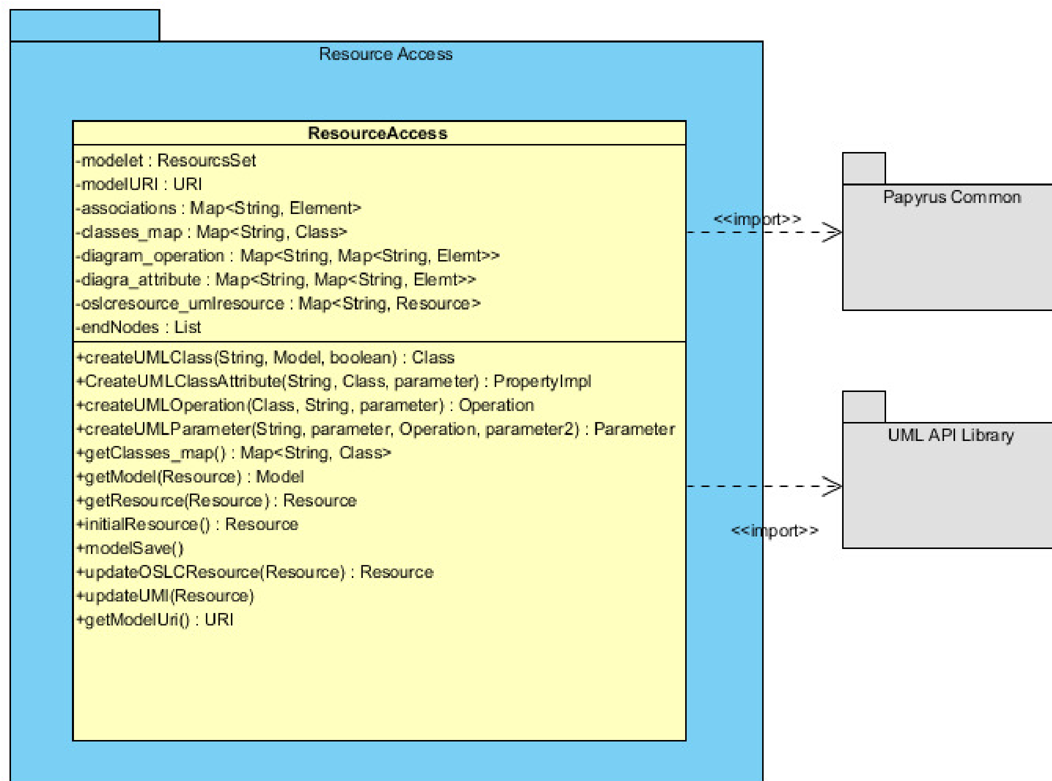

Abbildung 16 : Klassendiagramm von Resource Access

5.4.4.3. Komponete Persistence

Diese Komponente speichert alle OSLC Resourcen, welche im Payprus Service Provider enthalten sind. Dabei bietet sie auch alle nötigen Methoden für das Erzeugen, Löschen und Abrufen von OSLC Resourcen.
In dieser Komponete gibt es zwei Klassen. Die Klasse *Persistence* ist dafür zuständig die OSLC Resource zu speichern und bietet auch die Methode für das Erzeugen , Löschen einer OSCL Resource und dem Zugriff auf die Resource.
Die Klasse *Populate* ist eine Hilfeklasse, die intern von der Klasse *Persistence* benutzt wird. (Siehe Abbildung 17)

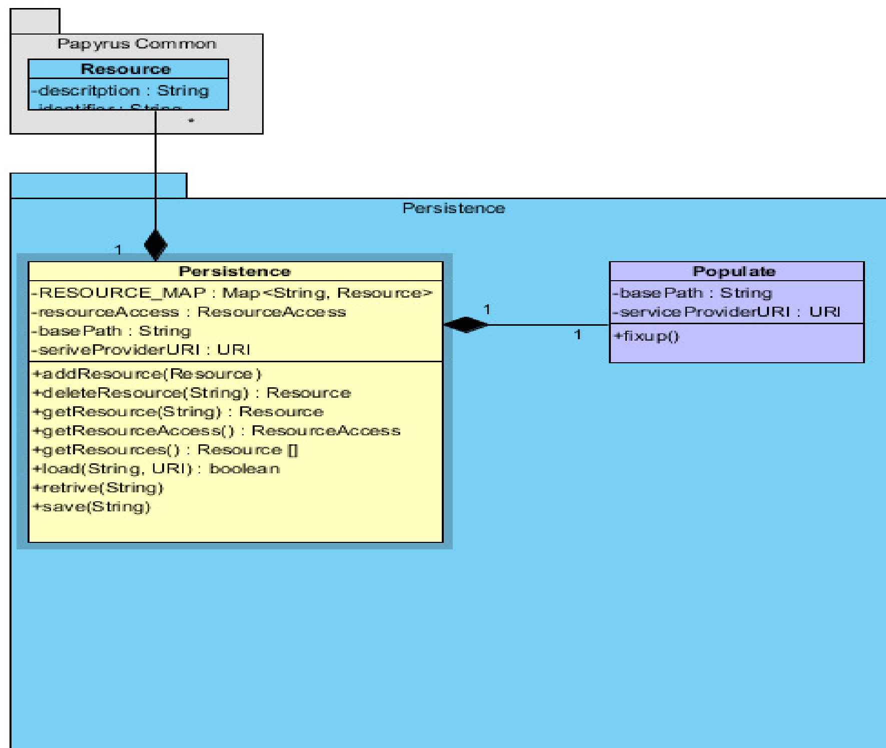

Abbildung 17 : Klassendiagramm von Persistence

5.4.4.4. Komponente OSLC Resource Service

Diese Komponente repräsentiert den Papyrus Provider und definiert auch die Services, die von dem Service Provider angeboten werden.

Die Klasse *PapyrusResourceService* definiert die Services, die von Service Provider via HTTP angeboten werden. Wie zum Beispiel die Behandlung von HTTP-Requests wie GET, CREATE, DELETE, UPDATE. Außerdem wird hier auch der Path der OSLC

Resourcen angegeben. Unter dem Path befinden sich alle OSCL Resourcen, die von dem gleichen Service Provider veröffentlicht werden .
Die Klasse *OSLC4JPapyrusApplication* kann hier als Repräsent von Service Provider angesehen werden. Hier wird dann eine JAX-RS Servlet Applikation anhand der Oberklasse *OslcWinkApplication* kreiert. Wie alle wichtigen Informationen wird die Klasse von der Resource, der Path vom Resourceshape[1] an *OslcWinkApplication* weitergegeben.

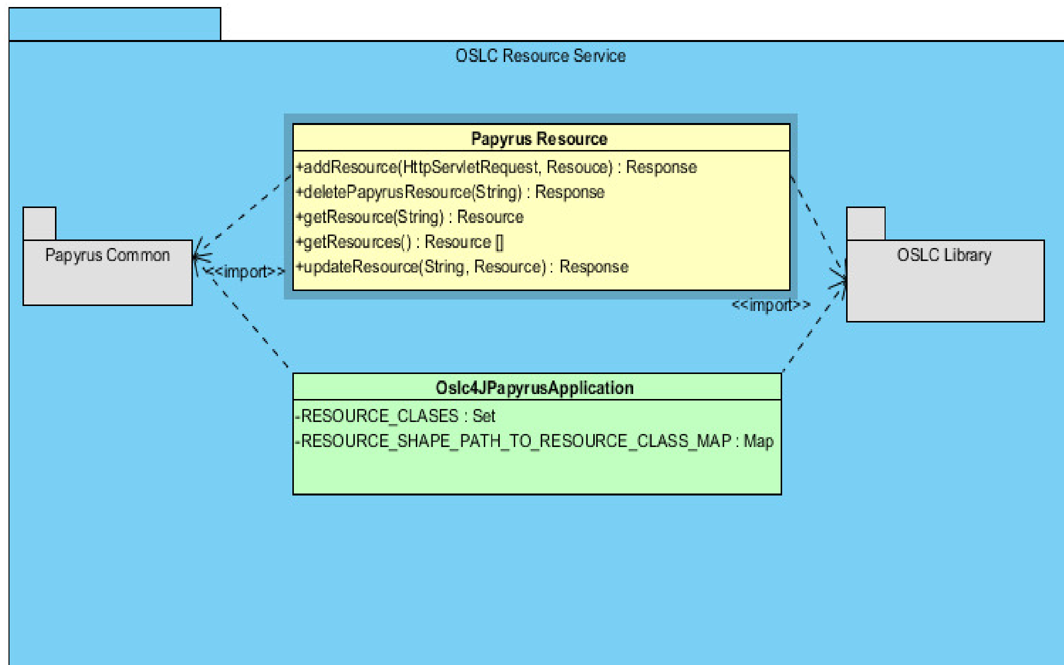

Abbildung 18 : Klassendiagramm von OSLC Resource Service

5.4.4.5. Komponente Servlet

Diese Komponente hat drei Klassen, der *ServletListener* wird bei dem Anlegen von dem Dynamische Web Projekt automatisch erzeugt. Diese Klasse ist dafür zuständig die Änderungsevents über dem Server Lebenzyklus wahrzunehmen. Wenn die Web Applikation startet, in diesem Fall der Papyrus Service Provider, wird dann der *ServletLinstener* erzeugt. Diese Klasse werden auch für den Path vom Service Provider erzeugt, die als Basispath für untergeordnet OSLC Resources gilt.
Die Klassen *ServiceProviderFactory* und *ServiceProviderSingelton* sind dafür verantwortlich, einen einmaligen Service Provider zu erzeugen.

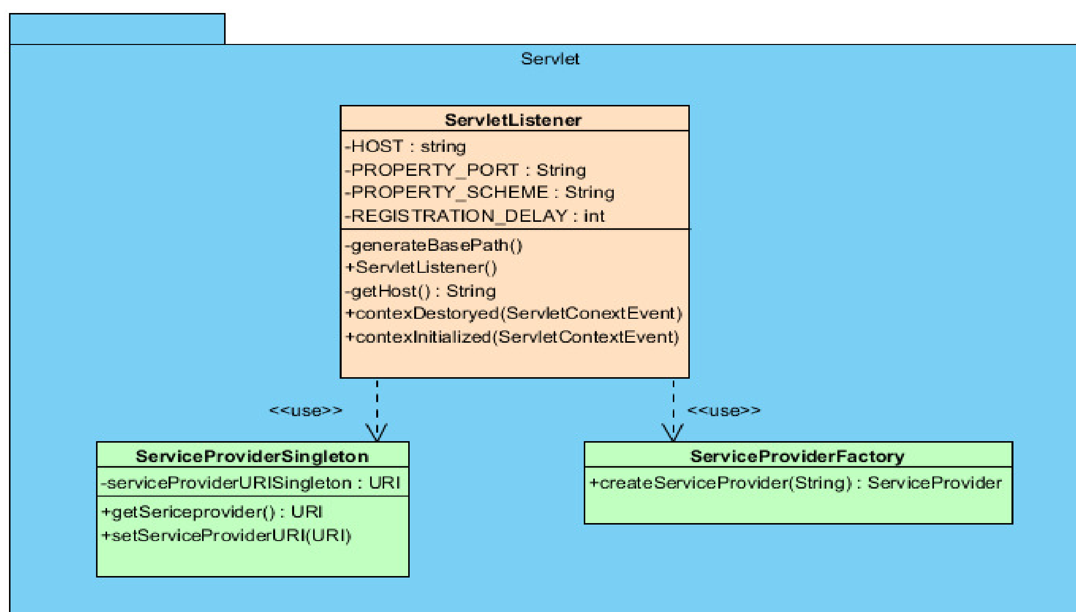

Abbildung 19 : Klassendiagramm von Servlet

5.4.5 Papyrus Consumer

Der Konsumer ist der Gegenspieler vom Provider. Jeder Provider braucht einen Konsumenten, um seine OSLC Resource verteilen zu können. Der Konsument ist dann in der Lage, die OSLC Resource via HTTP zu lesen, und in Klassenobjekten zu speichern. Dies ermöglicht dann die Bearbeitung mit heruntergeladene Information für weitere Zwecke.
Die Vorraussetzung dabei ist, dass der Konsument das gleiche Papyrus Common benutzt wie der Provider. Sonst könnte es zu Mappingfehler führen. Natürlich muss die OSLC 4J Library auch importiert werden.

5.4.5.1. Komponenten

Der Konsument hat nur eine Komponente, wie Abbildung 20 darstellt. Die Komponente ist abhängig von Papyrus Common und OSLC 4J Library.
Die Abildung 21 ist das Klassendiagramm. Die Klasse *Retriever* in dieser Komponente hat fünf Methoden.
Bei der Methode *addResource* wird eine neue OSLC Resource zu den vorhandenen Resourcen hinzugefügt, die Methode *deleteResource* tut das Gegenteil, hier wird eine von den vorhandenen OSLC Resourcen gelöscht. Bei *updateSystemdefinition* wird eine OSLC Resource von den vorhandenen aktualisiert. Als Rückgabewert wird

dann die *ClientResponse* zurückgegeben, um zu sehen, ob die Aufgabe der Methode erfolgreich durchgeführt wurde.
Die Methode *getQueryBase* ist eine Hilfsmethode für *getSystemDefinition*, und ist dafür zuständig , die OSLC Resourcen unter einer bestimmten URI zu holen.

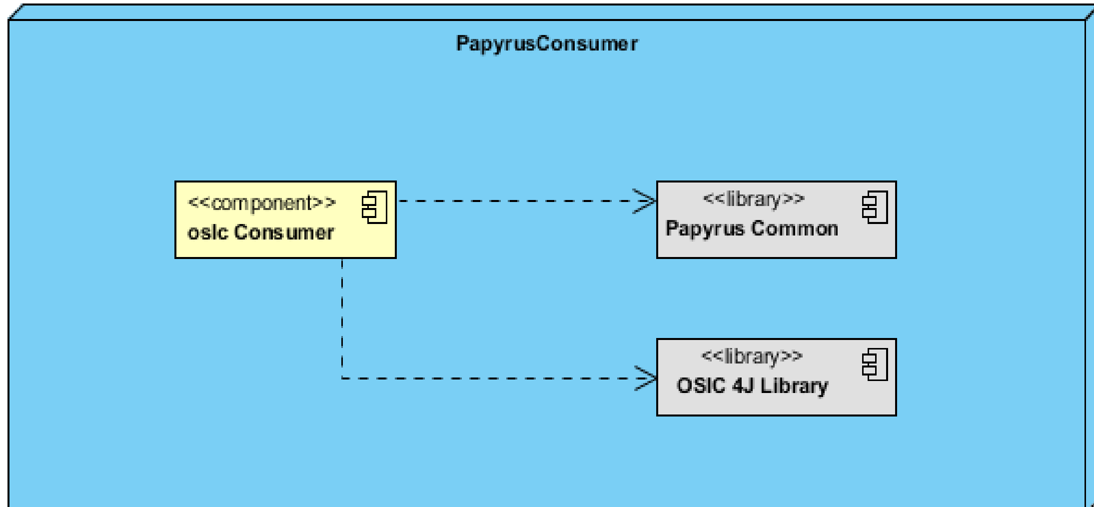

Abbildung 20 : PapyrusConsumer Komponenten

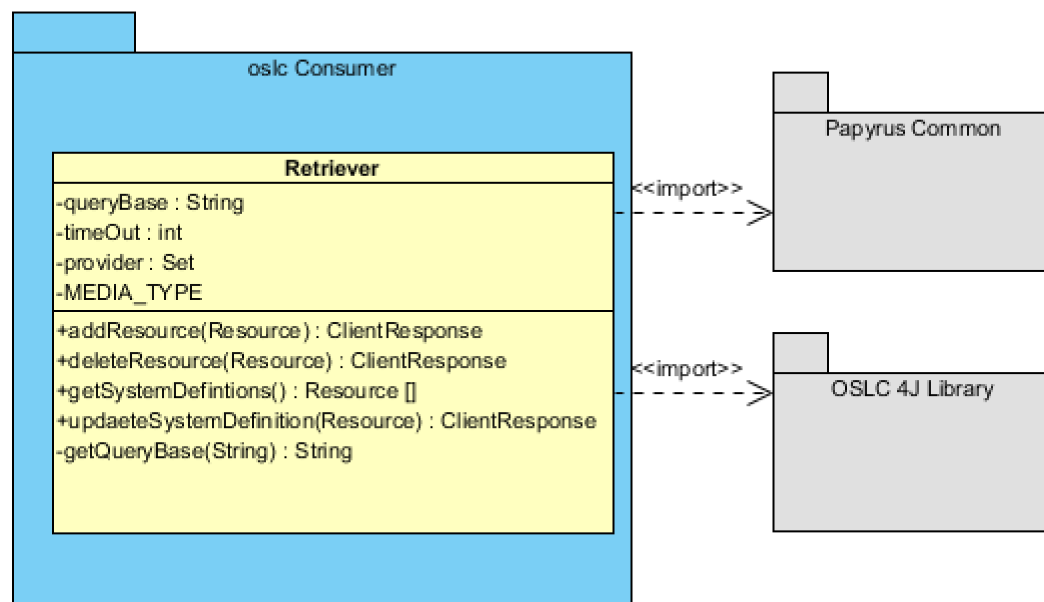

Abbildung 21 : Klassendiagramm von OSLC Consumer

Der Papyrus Consumer ist sehr adaptiv. Er kann von jedem importiert werden, der die OSLC Resource konsumieren möchten. Er bietet alle notwendigen Methoden für Lesen und updaten von OSLC Resourcen, was als Grundlage für Weiterentwicklung von Geschäftslogik auf der Clientseite sehr wichtig ist.

5.5 Implementation

Im diesem Kapitel wird die genaue Umsetzung von dem Entwurf des vorherigen Kapitels erläutert. Außerdem wird auch erläutert, was für Probleme es während der Entwicklung gab.

5.5.1 Konfiguration

Für die Entwicklung IDE wird hier Eclipse JUNO benutzt. Um OSLC4J [15] zu importieren gibt es zwei Varianten. Man kann Lyo SDK [16] herunterladen, und die Jar-Datei direkt importieren. Lyo SDK wurde von der OSCL Gemeinschaft für Eclipse entwicklt, es enthält alle OSLC Bibliotheken. Es ist der einfachst Weg, um OSLC in Eclipse zu Benutzen. Allerdings sollte man hier beachten, dass man die gleiche Version von SDK benutzt wie die vom Provider, wenn man später in anderen Tools den Konsumenten importiert.

Eine andere Alternative ist die Maven Abhängigkeit. Um dies durchzuführen, muss man zuerst das Lyo Projekt vom Git Repository auschecken. Die Git Repository Adresse wird auf der Lyo Homepage [18]angegeben. Weil die Lyo Projekte alles Maven Projekte sind, kann man dannach Auschecken und den Build der OSLC Abhängigkeit vom Lokale Maven Repository importieren. Es ist allerdings relativ unhandlich. Das Tutorial auf der OSLC Homepage wird allerdings auf diese Art und Weise aufgebaut. Für Anfänger ist das aber nicht unbedingt die optimale Wahl.

5.5.2 Probleme

Am Anfang der Entwicklung gab es einige Schwierigkeiten zu überwinden. Da man keine genaue Übersicht darüber hat, was man machen muss, auch wenn man das Tutorial auf der OSLC Homepage gelesen hat. Da der Prototyp in dieser Arbeit vor dem SDK Release gestartet wurde, musste die Maven Varainte benutzt werden. Allerdings waren die Lyo Projekte nicht fehlerfrei, allein um das Beispiel von Lyo laufen zu lassen, kostete es eine Menge Zeit und Energie. Nach dem Release von SDK wurde die Situtation dann erheblich besser. Die Online Dokumentation ist aber immer noch keine große Hilfe für das Starten einer Entwicklung. Man kann nicht aus

der Dokumentation schliessen, womit man anfangen soll um ein OSLC Project zu initialisieren.

Die einzigen Anhaltspunkte sind das Beispiel vom Lyo Projekt, der OSLC Stockquote und der Workshop mit Bugzillia. Von dem OSLC Stockquote kann man vermuten , wie das Programm geteilt und später implementiert werden soll. Allerdings ist das Beispiel nicht so geeignet für die Erläuterung der Funktionsweise von OSLC. Die Gründe hierfür sind, dass das Beispiel die Information aus dem Internet bezieht und nicht aus lokalen Quellen. Das bedeutet, dass die Funktionsweise vom Service Provider aus dem Beispiel nicht genau nachvollzogen werden konnte und in diesem Falle sogar in falscher Weise verstanden wurde.
In dem Beispiel wurde die OSLC Resource jedes Mal neu generiert, wenn HTTP GET aufgerufen wurden. Eigentlich sollte diese Aufgabe von einer ganz anderen Komponete im Service Provider ausgeführt werden. Auch die wesentlichen Funktionen wie ADD Update und Deletet wurden hier gar nicht benutzt, beziehungsweise es war nicht möglich diese zu benutzen. Wenn man also das Beispiel als Startpunkt nutzt, um zu sehen wie Service Provider funktioniert, wird man schnell frustriert sein.

Der Workshop mit Bugzilla ist ein wenig besser. Aber die Dokumentation ist eher für ein Seminar mit Moderator geeigent als zum Selbststudium, da in der Dokumentation nichts über den Adpater von Bugzilla erklärt wird. Am Ende man weiß vielleicht wie Update, ADD, Delete funktionieren, aber eine Übersicht des Projekts hat man immer noch nicht. Man weißt zum Beispiel überhaupt nicht wie OSLC Resourcen kreiert werden.
In OSLC werden viele Prefix, Annotation verwendet. Allerdings ist eine genaue Erklärung dazu bis jetzt nicht zu finden. Das erschwert auch das Starten von Projekten. Als das PapyrusCommmon geschrieben wurde, war es nicht einfach herauszufinden für welchen Datentypen welche Repräsentation in der Annotation benutzt werden sollte. Bis jetzt ist die Repräsentation für die Maps immer noch unklar.

5.5.3 PapyrusCommon Implementation

Wie es schon im vorherigem Kapitel aufgezeigt, ist das PayprusCommon einfach struktuiert. Der Vorteil dabei ist , dass man eine gute Übersicht hat was wohin gehört, und wie die Aufgabe der einzelnen Teile ist. Angelegt wird PapyrusCommon als Maven Projekt in Eclipse.

5.5.3.1. Klassen , Methoden

In der Komponente Papyrus Resource sind die Klassen so implementiert, dass wichtige UML Information von Papyrus aufgenommen werden können. Die Attribute in den Klassen können natürlich nach Bedarf erweitert werden. Die meisten Methoden in den Klassen sind Getter und Setter. Für jedes Attribut, welches in die OSLC Resource aufgenommen werden soll, müssen beiden Methode vorhanden sein ansonsten würde es zu Exceptions führen. Daher bleiben die Klassen schlank und einfach zu erweitern.

Während der Entwurfsphase sollten die Klassen mehr generisch gestaltet werden. Das heißt Interfaces oder abstrakten Klassen zu benutzen anstatt die Klassen direkt. Zum Beispiel die Klasse *Klassendiagramm* sollte einem Interface Diagramm vererben.
Allerding tauchte beim ausführen des Service Providers Mappingfehler auf. Es schien , dass OSLC die Informationen im Objekt vom Interface nicht der entsprechenden Klassen zuordnen kann. Vielleicht könnte dieses Problem durch ein bestimmter Annotation gelöst werden, es wurde aber nach langer Recherche leider keine Lösung gefunden. Aus diesem Grund muss hier *Diagramm* direkt als Klasse benutzt werden.
In der Komponente OSLC Resource sind die Klassen ähnlich gestaltet, hier werden die Informationen, die OSLC bezogen sind, aufgenommen. Zum Beispiel um die Verfolgbarkeit zu ermöglichen wurden die Klassen *RelatedAritfact* und *ChangeAction* eingefügt. Die beiden Klassen ermöglicht das Aufnehmen von Beziehung zwischen Artefakte und dem Dokumentieren von Änderungen. Außerdem werden die wichtigen Konstanten für den Namensraum und Prefixe für OSLC Resourcen defininiert. Die werden hauptsächlich in der Annotation verwendet.

5.5.3.2. Annotation

Die Implementierung der Klassen ist in PapyrusCommon nicht kompliziert. Die Annotation ist hier allerdings nicht auf einen Blick zu durchschauen. In diesem Abschnitt werden die verwendeten Annotationen erklärt.

```
@OslcNamespace(PConstants.PAPYRUS_NAMESPACE)
@OslcResourceShape(title = "FOAF Attributs Resource Shape", describes = PConstants.TYPE_ATTRIBUT)
public class Attribut {
```

Abbildung 22 : OSLC Annotaion vor Klassenimplementation

Für jede Klassen, die später in die OSLC Resource aufgenommen werden soll, muss der Namensraum für diese Klasse angegeben werden.

Wie in Abblidung 22 dargestellt, ist die Annotation *@OlscNamespace* dafür zuständig.

Anhand des Namensraums kann die OSLC Resource in unterschiedlichen Domänen geteilt werden, zum Beispiel die Domäne für Entwurf, oder für Change Mangement , so dass der Konsument nur die dazugehörigen Information auslesen muss, anstatt die ganzen Informationen vom Service Provider holen. Dem liegt zu Grunde, dass ein Service der OSLC Resource in unterschiedlichen Domänen veröffentlicht werden kann. In diesem Prototyp wird nur ein Namensraum benutzt.

Die Zeile *@OslcResourceShape* gibt eine kurze Beschreibung über die betroffene Klassen in dem Resource Shape. Resource Shape ist eine Art Sammlung von Elementen der OSLC Resource. Alle definierten Elemente der OSLC Resource können hier mit Beschreibung wieder gefunden werden. Genaue Informationen darüber findet man in der Core Spezifikation auf der Homepage [15].

```
@OslcDescription("Parameters of Operation ")
@OslcName("parameters")
@OslcPropertyDefinition(OslcConstants.DCTERMS_NAMESPACE + "parameters")
@OslcRepresentation(Representation.Inline)
@OslcValueType(ValueType.LocalResource)
@OslcTitle("parameters")
public List<OperationParameter> getParamters() {
    return parameters;
}
```

Abbildung 23 : OSLC Annotaion für List

Die Abbildung 23 zeigt wie eine Annotation für den Datentyp *List* ausieht. Die *@OslcDescription* gibt hier eine kurze Beschreibung für die betroffenen Attribute, in diesem Fall ist es das Attribut *paramenters* vom Datentyp *List.* Falls man mehr über die Attribute wissen möchte, kann man die Beschreibung im Resource Shape finden. Die *@OslcName* und *@Oslc*Title sind auch für die Darstellungs der Informationen der Elemente in ResourceShape zuständig.
Für die Darstellung der OSLC Resource an sich sind die *@OslcPropertyDefinition und @OslcRepresentation* sehr wichtig. Mit der *@OslcPropertyDefinition* wird das bezogene Element getagged. Die @OSLCRepresentation betimmt, wie das Element in OSLC repräsentiert werden soll. Daher wird *inline* mit der lokalen OSLC Resource zusammen benutzt.

@OslcValueType gibt hier an ob es eine Referenz oder Lokale OSLC Resource ist.

```
▼<dcterms:parameters>
  ▼<papyrus:OperationParameter>
      <papyrus:parameterType>return</papyrus:parameterType>
      <papyrus:datentype>String</papyrus:datentype>
      <papyrus:name>return</papyrus:name>
    </papyrus:OperationParameter>
  </dcterms:parameters>
```

Abbildung 24 : OSLC Resource Element Parameter im RDF /XML Format

Abbildung 24 zeigt, wie die OSLC Resource im RDF/XML Format vom Element *Parameters* in einer Methode aussieht.
Wie es in der Annotation schon definiert wurde, wurde das Attribut *parameters* hier als <dcterms:paramters> getagged. Die Elemente der List werde als untergeordnete Elemente von *parameters* dargestellt.

Neben den hier gelisteten Annotationen gibt es noch weitere Annoationen wie *@OslcReadOnly* oder *@OslcOccurs*. Mit diesen kann man festlegen, ob man die OSLC Resource aktualisieren kann, und wie oft das Element in dem Resource auftreten kann.

Auch wenn das PapyrusCommon an sich keine komplizierte Klassen oder Methode besitzt,beeinflusst die richtige Umsetzung der Annotationen direkt den gesamt Prototyp und vorallem ob er erfolgreich ausgeführt wird oder nicht.

5.5.4 Papyrus Provider Implemenation

Papyrus Provider ist ein dynamisches Web Maven Projekt. Mit dem dynamsichen Web Projekten hat man schon von Anfang an die Möglichekeit, einen Webservice zu starten. Das Projekt hat auch also ein Servlet Listener für die Handlung der JAX-Servlet.

Klassen und Methoden
In der Komponete OslcResourceService ist die Klasse *Oslc4JPapyrusApplication* dafür zuständig, dass die Resource mit OSLC Resource Shape Path zu mappen. Das heißt eigentlich , dass das Java Objekt hier als OSLC Resource gemappt wird. Außerdem werden alle Arte von Service Provider registriert.

```
static {
    RESOURCE_CLASSES.addAll(JenaProvidersRegistry.getProviders());
    RESOURCE_CLASSES.addAll(Json4JProvidersRegistry.getProviders());
    RESOURCE_CLASSES.add(ResourceService.class);

    RESOURCE_SHAPE_PATH_TO_RESOURCE_CLASS_MAP.put(Constants.PATH_RESOURCE, Resource.class);
}
```

Abbildung 25 : OSLC Resource Mapping

Wie in Abbildung 25 dargestellt, wird der Resourcen Path mit der Resourcen Klasse gemappt. Wenn man hier mehr als eine OSLC Resource mit Unterschiedlichen Service Providern veröffentlichen möchte, muss man an dieser Stelle entsprechend erweitern. Danach werden alle Maps an die Oberklasse weitergegeben. Die Oberklasse *OSLCWinkApplication* startet anhand diesen Information dann die JAX-RS Servlet Applikation.
Diese JAX-RS Servlet Applikation umfasst alle Leistungen , die vom Service Provider angeboten werden. Die Leistungen werden dann in der Klasse *PapyrusResourceService* definiert.

```
@OslcQueryCapability
(
    title = "Resources Query Capability",
    label = "Resources Catalog Query",
    resourceShape = OslcConstants.PATH_RESOURCE_SHAPES + "/" + PConstants.PATH_RESOURCE,
    resourceTypes = {PConstants.TYPE_RESOURCE},
    usages = {OslcConstants.OSLC_USAGE_DEFAULT}
)
```

Abbildung 26 : OSLC Query Capabilty

Die Abbildung 26 zeigt wie man ein *Query Capability* definiert. Hier wird der Path für eine Ansammlung aller OSLC Resources definiert. Wenn später die Konsumenten die OSLC Resources via HTTP-GET nachfragt, wird zuerst nach diesem Path gesucht.

```
▼<oslc:queryCapability>
  ▼<oslc:QueryCapability>
     <dcterms:title rdf:datatype="http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral"
     <oslc:resourceShape rdf:resource="http://Jovienn-PC:8080/PapyrusProvider/resourceSh
     <oslc:queryBase rdf:resource="http://Jovienn-PC:8080/PapyrusProvider/papyrusresourc
     <oslc:usage rdf:resource="http://open-services.net/ns/core#default"/>
     <oslc:resourceType rdf:resource="http://open-services.net/ns/cm#PapyrusResource"/>
     <oslc:label>Resources Catalog Query</oslc:label>
   </oslc:QueryCapability>
 </oslc:queryCapability>
```

Abbildung 27 : OSLC Resource Darstellung von Query Capability

Abbildug 27 zeigt, wie die Query Capbilty im OSLC Registry Katalog aussieht.
In der Zeile oslc:queryBase wird dann der Path für alle OSLC Resourcen angegeben.
Der OSLC Registry Katalog ist wie die gelben Seiten. Sie listet alle registrierten Services und Resourcen auf. Wenn man einen bestimmten Service Provider oder eine Resource finden möchtet, greift man hierauf zu.

Im *PapyrusResourceService* werden auch die Methoden. die für die Behandlung der HTTP Request zuständig sind definiert . Vor den Methoden werden die Annotationen wie *@GET* und *@POST* geschrieben.

```
@GET
@Path("{resourceId}")
@Produces({OslcMediaType.APPLICATION_RDF_XML, OslcMediaType.APPLICATION_XML, OslcMediaType.TEXT_XML,
public Resource getResource(@PathParam("resourceId") final String resourceId)
```

Abbildung 28 : Annotation für Behandelung von HTTP Request GET

In Abbildung 28 wird dargestellt, dass die Methode *getResource* auf HTTP GET nur für ein bestimmtes OSLC Resource reagiert. Hier gibt *@Path* dann die ID von der geforderten Resource zurück. *@Produces* gibt an, in was für einem Format die HTTP Response zurückgeschickt werden können.

5.5.4.1. Komponente Persistence

In der Komponente *Persistence* gibt es die Klassen *Persistence* und *Populate.* Die wesentlichen Aufgaben von den Beiden ist das Speichern von OSLC Resource Instanzen, die vom Service Provider umfasst werden. Ebenso bieten die Klassen auch die Methoden für den Zugriff auf und das Kreieren von OSLC Resourcen. Alle OSLC Resource Instanzen sind hier vom Datentyp Map. Der Schlüssel der Map ist die URI der OSLC Resource.

5.5.4.2. Komponete ResourceAccess

Die Klasse *ResourceAccess* ist für den Zugriff auf die Uml Datei verantwortlich. Um das erst zuermöglichen braucht die Klasse folgende Bibliotheken.

Bibliothek	Version
org.eclipse.emf.ecore.xim	2.8.1
org.eclipse.emf.ecore	2.8.3
org.eclipse.uml2.common	1.7.0
org.eclipse.uml2.type	1.0.0
org.eclispe.emf.common	2.8.0
org.eclipse.uml2.uml	4.0.2

Tabelle 1: Bibliothek für die Klasse *ResourceAccess*

Für das Erstellen von OSLC Resourcen spielt die Methode *getResource* eine wichtige Rolle. In der Methode wird die UML Datei aus dem Filesystem gelesen. Als Paramter wird hier einen Instanz vom Typ *Resource* übergeben. Nach dem Lesen werden die gesamten Informationen von UML in der Instanz vom *Model* gespeichert (Siehe Abbildung 26).

```
Model model = (Model) EcoreUtil.getObjectByType(
        modelResource.getContents(), UMLPackage.Literals.MODEL);
```

Abbildung 26

Um auf den Inhalt zuzugreifen, kann man hier einen Iterator benutzen. Dies funktioniert weil in dem Iterator die Instanzen alle als *Eobject* zurückgegeben werden, hier muss man den entsprechenden Typ casten, um die Daten richtig auslesen zu können.
Die gelesenen Informationen werden auch in der übergebene Insantz von *Resource* gespeichert.
Außerdem werden sie auch in verschiedenen Maps gespeichert um später einfach zurück auf die Instanze in *model* zugreifen zu können.
Die folgende Tabelle ist eine Übersicht über die genutzten Maps.

Map	Beschreibung
Map<String,Class>classes_map	Lokale Variable in der Klasse *ResourceAccess.* Benutzt den Name des Klassendiagramms als Schlüssel, gespeichert wird die Class Instanz vom gelesenem UML Model
Map<String, Element> attributeuml	Lokale Variable in der Methode *getResource.* Speichert die Attribute der Instanz innerhalb einer Klasse vom UML Model. Der Schlüssel hier ist der Name der Attribute.
Map<String, Element> operationMap	Lokale Variable in der Methode *getResource*. Speichert die Instanz von *OperationImp* innerhalb einer Klasse vom UML Modell. Der Name und der Rückgabetyp sind im Format String die Schlüssel der Map.
Map<String, Map<String, Element>> diagram_operation	Lokale Variable in der Klasse *ResourceAccess.* Hier werden die Methoden mit den Klassen gemappt. Schlüssel ist der Name der Klassen, die *operationMap* wird in dieser Map gespeichert.
Map<String, Map<String, Element>> diagram_attribute	Lokale Variable in der Klasse *ResourceAccess.* Hier werden die Attribute mit der Klassen gemappt. Schlüssel ist der Name der Klassen, die attributuml wird in dieser Map gespeichert
Map<String, Element> associations	Lokale Variable in der Klasse *ResourceAccess.* Speichert die Instanz von der Asscoiation in dem UML Model. Schlüssel ist der Name der Asscociation.
Map<String, List<Element>> associationEnd_Element	Lokale Variable in der Klasse *ResourceAccess.* Jede Asscociation hat zwei Enden. Jedes Ende hat eine Reihe von Eigenschaften. Diese Eigenschaften werden in der List gespeichert, und dann wieder in der Map. Der Schlüssel ist der Namer des Endes.

Tabelle 2: Map für Speicherung von eingelesene Information aus UML Datei

Wenn die *getResource* Methode für das Lesen von Information ist, so gibt es die Methode für die Aktualisierung der OSLC Resource und Aktualisierung der UML Datei.
In der Methode *updateOSLCResource* werden erneut die Information von der UML Datei ausgelesen. Anschließend werden die neue Inforamtionen mit den alten Informationen der OSLC Resource verglichen. Hier werden auch Änderung übermittelt und dokumentiert. Zum Schluss wird die alte OSLC Resource durch die neue Resource ersetzt.
Zum Vergleich wird hier die Methode *compareOSLCResource* aufgerufen. In der Methode werden alle Elemente vom UML Model und der OSLC Resource anhand der Namen Methdoe zusätzlich auch mit Typen miteinander verglichen. Falls ein Element vom UML Model in der OSLC Resource nicht gefunden wird, dann heißt dies das ein neues Element hinzufügt wird. Umgekehrt bedeutet es, dass ein Element gelöscht wird.

Für die Aktualisierung der UML Datei wird die Information von der OSLC Resource in die UML Datei geschrieben. Wenn man die Methode für die Aktualisierung aufruft bedeutet das, dass die Informationen der OSLC Resource direkt von der UML Datei übernommen werden soll. Deshalb muss man hier keinen aufwändigen Vergleich betreiben. Man kann direkt die alte Information durchd die Neue ersetzen. Die *ResouceAccess* Komponente wird auch in dem *PapyrusCommon* Projekt kopiert, damit später der Client auch direkten Zugriff auf die Funktionalität für Aktualisierung der Resource erlangen kann. Eigentlich kann man nur eine *ResourceAccess* Komponente insgesamt haben, da wenn man die Komponente aus *PapyrusCommon* in dem Provider benutzt, es zu einer *NullPointerException* führt. Hier wird ein Einbindungsfehler der Bilbliothek zur Laufzeit des Providers vermutet. Nach langen Versuchen wurde hier leider keine Lösung gefunden.

Komponente Servlet
Diese Komponente ist dafür zuständig einen einmaligen Service Provider zu erzeugen und die Events des Lebenszyklus der OSLC Resource zu behandeln.
In der Klasse *ServiceProviderFactory* wird der Service Provider mit den Services, die in der Klasse *PapyrursResourceService* definiert sind, initialisiert.(Sehe Abbildung 29)

```
final ServiceProvider serviceProvider = org.eclipse.lyo.oslc4j.core.model.ServiceProviderFactory.
        createServiceProvider(baseURI,ServiceProviderRegistryURIs.getUIURI(),
                              "OSLC Papyrus Service Provider",
                              "Reference Implementation OSLC Papyrus Service Provider",
                              new Publisher("Heng Cao", "HAW"),
                              RESOURCE_CLASSES
```

Abbildung 29 : Instanzieren von Service Provider

Die Prefix Definitionen werden anschließend in dem Service Provider gesetzt. Somit werden auch die Namensräume definiert.(Sehe Abbildung 30)

```
final PrefixDefinition[] prefixDefinitions =
{
    new PrefixDefinition(OslcConstants.DCTERMS_NAMESPACE_PREFIX,   new URI(OslcConstants.DCTERMS_NAMESPACE)),
    new PrefixDefinition(OslcConstants.OSLC_CORE_NAMESPACE_PREFIX, new URI(OslcConstants.OSLC_CORE_NAMESPACE)),
    new PrefixDefinition(OslcConstants.RDF_NAMESPACE_PREFIX,       new URI(OslcConstants.RDF_NAMESPACE)),
    new PrefixDefinition(OslcConstants.RDFS_NAMESPACE_PREFIX,      new URI(OslcConstants.RDFS_NAMESPACE)),
    new PrefixDefinition(PConstants.CHANGE_MANAGEMENT_PREFIX,   new URI(PConstants.CHANGE_MANAGEMENT_NAMESPACE)),
    new PrefixDefinition(PConstants.PAPYRUS_NAMESPACE_PREFIX,   new URI(PConstants.PAPYRUS_NAMESPACE)),
};

serviceProvider.setPrefixDefinitions(prefixDefinitions);
```

Abbildung 30 : Prefix Definition in Service Provider

Die Klasse *ServletListener* implementiert das Interface zum Abfang der Events der Änderung .

In der Methode *run* wird dann die Resource aus UML Datei gelesen. Dafür wird die Klasse *Persistence* benutzt.

5.5.5 PapyrusConsumer Implementation

5.5.5.1. Klassen und Methoden

Die Aufgabe des PapyrusConsumer ist sehr einfach und bietet alle notwendigen Methoden für die HTTP- Request damit das C.R.U.D Prinzip erfüllt wird. Mehr machen auch die Consumer nicht, denn er ist nicht für Geschäftslogik auf dem Client zuständig. Je nachdem wer den Consumer benutzt, hat man dann unterschiedliche Anwendungen für die erlangten OSLC Resource Informationen.

Für das Auslesen der OSLC Resource muss man hier zuerst die richtige URI und Domain haben. Dafür sorgt die Methode *getQueryBase*.

Zuerst werden alle Service Provider aus dem Registry Katalog gelesen. In den Service Providern wird dann nach der gewünschten Domäne gesucht, zum Schluss nach dem Typ der Resource. So kann man dann Schritt für Schritt die OSLC Resource finden. Siehe Abbildung 31

```
String domain = PConstants.CHANGE_MANAGEMENT_DOMAIN;
String type = PConstants.TYPE_RESOURCE;

final ServiceProvider[] serviceProviders = new ServiceProviderRegistryClient(providers, MEDIA_TYPE,
        registryUri).getServiceProviders(); // (PROVIDERS, mediaType).getServiceProviders();

if (serviceProviders != null) {
    for (final ServiceProvider serviceProvider : serviceProviders) {
        final Service[] services = serviceProvider.getServices();

        for (final Service service : services) {
            if (domain.equals(String.valueOf(service.getDomain()))) {
                final QueryCapability[] queryCapabilities = service.getQueryCapabilities();

                for (final QueryCapability queryCapability : queryCapabilities) {
                    final URI[] resourceTypes = queryCapability.getResourceTypes();

                    for (final URI resourceType : resourceTypes) {
                        if (resourceType.toString().equals(type)) {
                            return queryCapability.getQueryBase().toString();
```

Abbildung 31: Methode *getQueryBase*

Wie in Abbildung 30 dargestellt, ist man dazu in der Lage auf die OSLC Resource zuzugreifen, nachdem die QueryBase gelesen wurde. Hier muss man nur die schon von OSLC angebotenen Methode *getOslcResources* benutzen.

```
public final Resource[] getSystemDefinitions() throws ConsumerException {
    try {
        final OslcRestClient oslcRestClient = new OslcRestClient(providers, queryBase, MEDIA_TYPE, timeout);

        final Resource[] papyrusSystemDefinitions = oslcRestClient.getOslcResources(Resource[].class);
```

Abbildung 32: Aufruf von OSLC Rest Methode

Für das Erzeugen, Aktualisieren, und Löschen von OSLC Resourcen sind die Methoden ähnlich.
OSLC bietet alle REST-fähige Methoden, um alle diese Aktionen zu erfüllen.
Abbildung 33 zeigt als Beispiel die Methode zum Aktualisieren.

```
public final ClientResponse updateSystemDefinition(final Resource updatedResource) {
    String resourceBase = queryBase + "/" + updatedResource.getIdentifier();
    final OslcRestClient oslcRestClient = new OslcRestClient(providers, resourceBase, MEDIA_TYPE, timeout);
    final ClientResponse clientResponse = oslcRestClient
            .updateOslcResourceReturnClientResponse(updatedResource);
    return clientResponse;
}
```

Abbildung 33: Methode für Aktualisieren von OSLC Resource

Hier wird auch die ID von der Zielresource angegeben. Sie wird ebenso anhand der Annotation *@Path("{resourceId}")* in der Klasse *PapyrusResourceService* abgefangen.

Der Client, der den Consumer importiert hat, kann dann sofort die Information aus der OSLC Resource lesen.

5.6 Test

Um den Prototype zu testen wurden mehrere Testszenarien drüchgeführt. In diesen Szenarien gibt es zwei Clients, die auf unterschiedliche UML Datein zugreifen. Die zwei Clients haben nur jeweils zwei Funktionalitäten, nämlich die OSlC Resource zu lesen und in die UML zu schreiben, und die OSLC Resource zu aktualisieren.

Testszenarios	Erwartetes Resultat
Startet Service Provider. Erstellt ein neues Klassen Diagramm in UML A Aktualisieren die OSLC Resource	In der OSLC Resource wird dann die Änderung Dokumentiert
Client für UML B ruft *get* Methode auf	Die neue Klasse soll in UML B erstellt werden.
Fügt neue Attribute und Methode in der neuen Klasse in UML B, und startet dann dessen Client für Aktualisierung der OSLC Resource.	Das OSLC Resource wird entsprechend aktualisiert. Alle Änderung werden notiert (Siehe Abbilung 36). UML A wird automatisch auch aktualisiert.
Neue Association und Generalization zwischen zwei Klassen hinzufügen in UML A, benutzen des Client zum Aktualisieren. Client für UML B ruft die Methode auf.	Die OSLC Resource wird entsprechend aktualisiert, und die UML B soll die gleiche Änderung haben.
Stress Test: Mehrere Klassen Attribute, Methode, Association und Genarlization in eine UML Datei hinzufügen, und auch Attriubte, Methode löschen. Starten des Clients für Aktualisierung. Der andere Client startet mit der *get* Methode	Alle Änderungen werden in der OSLC Resource angezeigt. Die UML Datei nach dem Aufruf von *get* muss entsprechend aktualisiert werden

Tabelle 3: Test für den Prototyp

In Abbildung 34 wird die im Test verwendete UML dargestellt. Die entsprechende Grafikdarstellung ist in Abbildung 35 dargestellt.

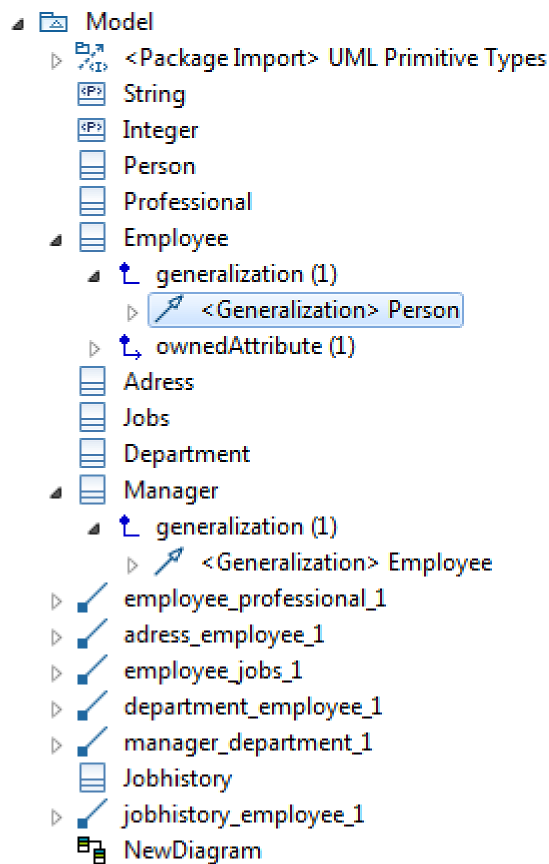

Abbildung 34: UML Struktur

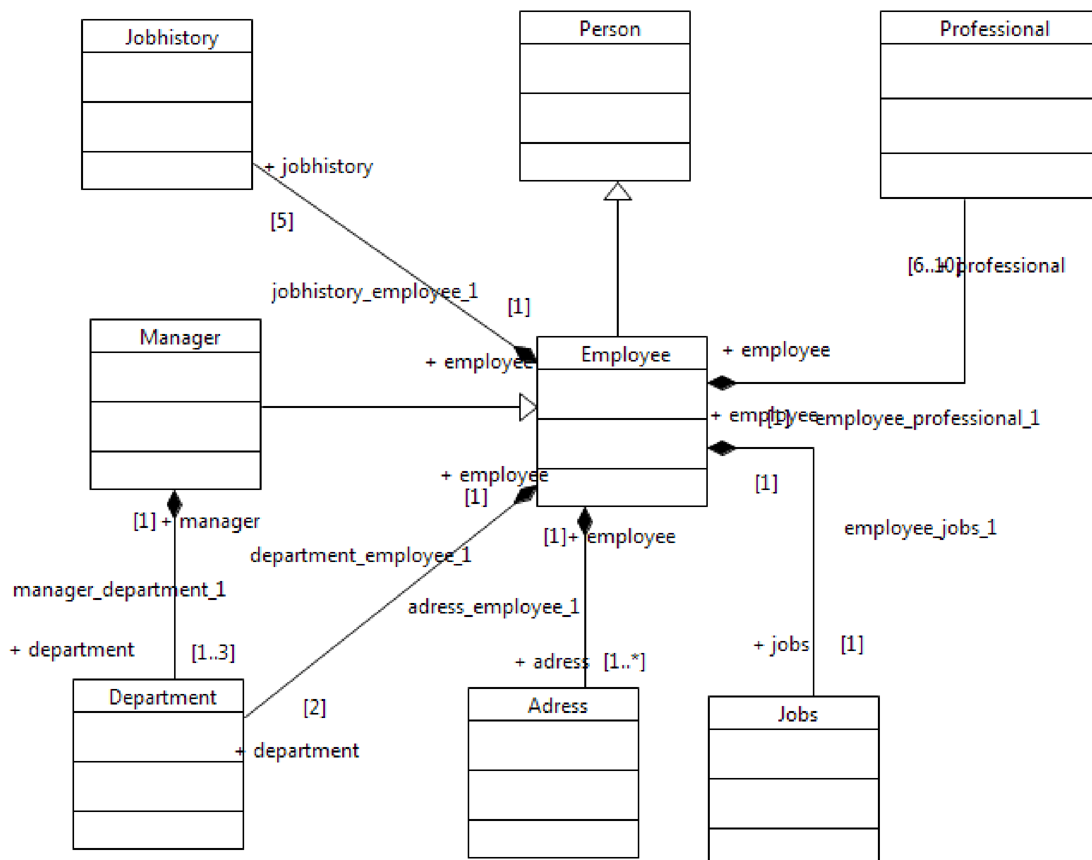

Abbildung 35: Papyrus Grafikdarstellung von UML

```
▼<dcterms:action>
  ▼<papyrus:ChangeAction>
     <papyrus:bezeichnung>ATT: name in Person ADDED</papyrus:bezeichnung>
   </papyrus:ChangeAction>
 </dcterms:action>
 <dcterms:updateDate rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2013-
 <dcterms:source rdf:resource="file:///E:/HAW/workspace2/PapyrusModel/model.uml"/>
```

Abbildung 36: Dokumentierung von Änderung und Beziehung

In diesen Testszanrien repräsentieren UML A und UML B unterschiedlichen Artefakte.

OSLC bietet die Möglichkeit, Beziehung zwischen beiden Artefakte aufzubauen. Das ist die Vorausetzung der Verfolgbarkeit. Außerdem wird auch die Änderung der OSLC Resource und die Zeit der Änderung notiert. Auch wenn sie unterschiedliche Artefakte wären, kann leicht von eine Artefakt zum anderen verfolgt werden.

Alle Tests ergaben die erwarteten Resultate. Anhand dessen wird gezeigt, auch wenn die beiden Artefakte keine direkte Verbindung miteinander haben, dass mit OSLC als Medium werden die Kommunikation zwischen Artefakt ermöglicht wird. Beiden können dann auch miteinander zusammen arbeiten. Hierbei kann auch das Testszaniro als Beweise für Erfüllbarkeit des Konzept der Verfolgbarkeit und Interoperabilität angesehen werden.

6 Zusammenfassung

Verfolgbarkeit und Interoperabilität bringen viele Vorteile in der Softwareentwicklung. In großen Softwareprojekten können dann alle Artefakte von Entwurf bis Test verfolgt werden. Man kann mit Hilfe dessen zum Beispiel sehen ob eine Anforderung erfüllt wurde. Seit langer Zeit hat man versucht, eine Lösung zu finden. Aber bis jetzt verursachten solche Anstrengungen großen Aufwand und Kosten.
OSLC bietet hier als Lösung für Verfolgbarkeit und Interoperabilität umfangreiche Funktionalität und Möglichkeiten der Gestaltung von individuellen Lösungen. Ein weiterer Vorteil von OSLC ist , dass man für jedes Tool nur einmal den Konsument und den Provider entwickeln muss. Im Vergleich dazu musste man früher immer neue Adapter für alle vorhandene Tools entwicklen, wenn ein neues Tool integriert werden musste. Nutzung von standard HTTP Protokoll, REST und RDF/XML Format erleichtern ebenfalls den Aufwand der Integration. Dadurch werden ebenfalls die Kosten reduziert.
Allerdings fehlen zu der Zeit dieser Arbeit noch passende Dokumentationen und Tutorials, um mit OSLC wirklich einzusteigen. Dies erschwert leider die Verbreitung von OSLC.
In dieser Arbeit wurde ein Prototyp für das Modelierungstool Papyrus erstellt. Dabei sollte die Verfolgbarkeit und Interoperabilität von OSLC getestet werden. Wenn man genau weiß, was getan werden muss, ist die Implementierung sehr schnell fertiggestellt. Dies ist ja genau ein Vorteil von der Strukturierung von OSLC. Die Aufteilung von Common, Provider und Consumer hält das gesamte Projekt übersichtlich. Dies ist wichtig für die Unternehmen, welche OSLC für vorhandene Tools benutzen wollen. Schnelle Integration von OSLC bedeutet auch weniger Kosten, um Verfolgbarkeit und Interoperabilität zu etabilieren.
Nach der Implementierung können beiden UML Datei miteinander zusammen arbeiten, obwohl keine direkte Verbindung zwischen ihnen exisitiert. Die OSLC Resource dokumentiert die Beziehung beider UML Dateien sowie alle Änderungen zwischen den Beiden.
Als Beispiel für Verfolgbarkeit ist es nicht optimal, da die Beiden nicht von unterschiedlichen Domains entstanden sind. Es kann daher nicht eindeutig gesehen werden , wie Artefakte durch den Lebenszyklus verfolgt werden. Aber es zeigt, wie

URI hier als Beziehung zwischen Artefakte dient und wie man leicht von einem Artefakte zu einem anderen gelangen kann. Es ist also kein Problem Artefakte über Domains hinaus zu verfolgen.
Für Verfolgbarkeit und Interoperabilität hat OSLC daher großes Potential, auch wenn es zur Zeit noch an der Dokumentation schwächelt. Aber mit Unterstützung von Firmen wie IBM hat es gute Möglichkeiten sich durchzusetzen.

7 Quellenangabe

1. http://www.w3.org/DesignIssues/LinkedData.html
2.http://linkeddatabook.com/editions/1.0/
3.http://www.linkeddatatools.com/introducing-rdf-part-2
4.http://www.w3.org/DesignIssues/RDF-XML.html
5. IEEE, Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice
6.IEEE, Model-Based Traceability
7.IEEE, Ontology-enhanced description of traceability services
8.http://www.praxiom.com/iso-definition.htm#Traceability
9.http://swt.cs.tu-berlin.de/lehre/seminar/ss07/folien/Traceability_Gruppe8.pdf
10.http://open-services.net/resources/presentations/w3c-linked-data-platform-and-oslc-v3/
11.http://www.w3.org/2001/sw/sweo/public/UseCases/IBM/
12.http://open-services.net/bin/view/Main/OslcCoreSpecification
13.http://open-services.net/resources/tutorials/integrating-products-with-oslc/
14.http://open-services.net/resources/tutorials/integrating-products-with-oslc/integrating-with-an-oslc-provider/sample-use-cases-for-oslc-consumers/
15.http://wiki.eclipse.org/Lyo/OSLCWorkshop
16.http://eclipse.org/lyo/download.php
17.http://wiki.eclipse.org/Lyo/LyoOSLC4J
18.http://eclipse.org/lyo/
19.http://de.slideshare.net/tgau/oslc-in-aktion
20.http://open-services.net/resources/videos/

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _________________ ______________________________