

Technische Universität Berlin  
Fakultät Elektrotechnik und Informatik  
Institut für Telekommunikationssysteme  
Fachgebiet Kommunikations- und Betriebssysteme



# Integrierte Entwicklungsprozesse

## Projektbericht zum Praxisprojekt Anwendungssysteme

31. März 2015

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
<b>2 Ausgangssituation</b>	<b>5</b>
<b>3 Zielvereinbarungen</b>	<b>5</b>
3.1 Evaluierung alternativer Implementierung . . . . .	5
3.2 Entscheidung für Ontologie . . . . .	8
3.3 Ziele für die Ontologie . . . . .	8
<b>4 Vorgehensweise bei der Modellierung</b>	<b>8</b>
4.1 Umsetzung . . . . .	8
4.2 Protégé-Fenster . . . . .	9
4.2.1 Class hierarchy-View . . . . .	9
4.2.2 Description-View . . . . .	9
4.2.3 Annotation-View . . . . .	10
4.2.4 Usage-View . . . . .	10
4.2.5 OntoGraf . . . . .	10
4.2.6 SPARQL Query . . . . .	10
4.3 Erstellung eines Ontologiegrundgerüsts . . . . .	10
4.4 Dokumente und Prozesse verfeinern . . . . .	14
4.5 Reasoner und Cardinalities . . . . .	17
4.6 Lernprozess . . . . .	20
4.6.1 Dokumente hinzufügen . . . . .	20
4.6.2 Prozesse hinzufügen . . . . .	20
4.6.3 Data Property hinzufügen . . . . .	20
4.6.4 Object Property hinzufügen . . . . .	21
4.6.5 Individuals erstellen . . . . .	21
4.7 Modellstatistik . . . . .	22
<b>5 formale Grundlagen der Object Properties</b>	<b>23</b>
5.1 inverse Beziehung . . . . .	23
5.1.1 Definition . . . . .	23
5.1.2 Beispiel . . . . .	23
5.1.3 Anwendung in Protégé . . . . .	24
5.2 transitive Beziehung . . . . .	27
5.2.1 Definition . . . . .	27
5.2.2 Beispiel . . . . .	27
5.2.3 Anwendung in Protégé . . . . .	28

<b>6 Beispiel: Frosti, der Kühlschrank</b>	<b>31</b>
6.1 Vorbereitung . . . . .	31
6.2 Durchführung . . . . .	31
6.3 Beispielablauf Produkt- und Geschäftsmodellentwicklung . . . . .	32
6.4 <i>SPARQL</i> -Beispiel . . . . .	36
<b>7 Herausforderungen in Protégé</b>	<b>38</b>
7.1 Modellierung der Objekteigenschaften . . . . .	39
7.2 Reasoner für die Modellüberprüfung . . . . .	40
7.3 Visualisierung und Abfrage mit OntoGraf und SPARQL . . . . .	42
7.4 Kompatibilität des Dateiformats OWL . . . . .	43
7.5 Stabilität des Modellierers und der Plugins . . . . .	43
<b>8 RESTful Web Services</b>	<b>44</b>
8.1 Eigenschaften von REST . . . . .	44
8.2 Python . . . . .	45
8.2.1 Vorteile . . . . .	45
8.2.2 Nachteile . . . . .	45
8.2.3 Code Obfuscation . . . . .	46
<b>9 REST-Beispiel in Python</b>	<b>46</b>
9.1 Anforderungen an die Umgebung . . . . .	47
9.2 Koordination mit Bugzilla-Server . . . . .	47
9.3 Python-Skript . . . . .	48
9.3.1 GET-Operation - Einen Bug einlesen . . . . .	48
9.3.2 POST-Operation - Ein Bug-Ticket erzeugen . . . . .	50
9.3.3 PUT-Information verändern . . . . .	51
9.3.4 POST und PUT . . . . .	51
<b>10 Open Services for Lifecycle Collaboration</b>	<b>52</b>
10.1 Heterogene Strukturen in Organisationen . . . . .	53
10.2 Ansatz: einheitliche Schnittstellen . . . . .	53
10.3 Reduzierter Implementierungsaufwand . . . . .	54
10.4 Vorbild Internet . . . . .	54
10.5 Adressierbarkeit von Ressourcen . . . . .	54
10.6 Services zur Verwaltung der Ressourcen . . . . .	55
10.7 Gemeinsamer Namensraum . . . . .	55
10.8 Basisoperationen CRUD . . . . .	55

<b>11 Umsetzung mittels Lyo/OSLC</b>	<b>57</b>
11.1 Funktionsweise und Grundstruktur . . . . .	58
11.2 Verbindungsaufbau und Abfrage . . . . .	58
11.3 Verbindungskomplexität zwischen REST-Anwendungen . . . . .	59
11.4 Vorschaufunktion für Bugzilla-Defekte . . . . .	60
11.4.1 Vorschau per REST-Client . . . . .	60
11.4.2 Vorschau per ninaCRM . . . . .	61
11.4.3 Diensterkennung . . . . .	65
<b>12 Perspektiven für ein Anschlussprojekt</b>	<b>65</b>
12.1 Schlusswort . . . . .	68
<b>13 Glossar</b>	<b>70</b>
<b>14 Namenskonventionen</b>	<b>72</b>
14.1 Abkürzungsverzeichnis . . . . .	72
14.1.1 Prozesse . . . . .	72
14.1.2 Dokumente . . . . .	72
<b>15 Klassendokumentation mit OWLDoc</b>	<b>73</b>
<b>16 Anhang</b>	<b>76</b>
<b>17 Implementierungshandbuch</b>	<b>77</b>
17.1 Lyo-/OSLC-Framework . . . . .	77
17.1.1 Voraussetzungen für die Inbetriebnahme . . . . .	77
17.1.2 Konfiguration für das Projekt . . . . .	78
17.1.3 Einrichtung in Eclipse . . . . .	78
<b>18 Matrix</b>	<b>80</b>

# 1 Einleitung

In der heutigen Zeit ist es für ein Unternehmen essentiell, auf Marktanforderungen innovativ und flexibel reagieren zu können. Um seine Überlebensfähigkeit zu sichern, muss es in der Lage sein, neue Kundenanforderungen schnell zu erkennen und umzusetzen. Eine besondere Bedeutung erhält diese Fähigkeit in der Konkurrenzsituation zu agilen und dynamisch aufgebauten Start-up-Unternehmen. Aufgrund dieser Anforderungen wird ein angemessenes Wissensmanagementsystem benötigt. Um reibungslose Unternehmensprozesse zu ermöglichen, ist eine unternehmensweit standardisierte Wissensrepräsentation und -distribution notwendig. Besondere Bedeutung erhält dieses Kriterium in einem Innovationsprozess, welcher eine fortlaufende Kommunikation über den gesamten Prozessverlauf benötigt, um somit eine lückenlose Datenintegration vom Geschäftsmodell bis zur Produktentwicklung zu ermöglichen.

In diesem Projekt wurde eine Ontologie erstellt, welche zur Repräsentation der Daten innerhalb eines Innovationsprozesses dient. Dafür wurde mit Hilfe des Ontologie-Editors Protégé und der Vorlage für ein agiles Produkt- und Geschäftsmodell eine erweiterbare Wissensrepräsentation entwickelt. Diese soll nicht die detaillierte Abbildung sämtlicher Geschäftsabläufe darstellen. Vielmehr dient sie dazu, eine vereinheitlichte Darstellung der Unternehmensdaten und ihrer Bedeutung sowie den prozessübergreifenden Zugriff auf diese zu ermöglichen. Durch SPARQL-Abfragen und die grafische Darstellung innerhalb eines OntoGrafen wird die Korrektheit der Ontologie verifiziert. Um auch einen anwendungsübergreifenden Zugriff auf die Daten zu ermöglichen, wurde mit Hilfe von REST und dem Lyo-/OSLC-Framework ein Prototyp entwickelt, der diesen Austausch ermöglicht.

In diesem Dokument wird das Vorgehen bei der Entwicklung der Ontologie und der REST- und OSLC-basierten Anwendung beschrieben. Es gliedert sich in zwei Hauptteile.

Der erste Teil befasst sich mit der Ontologie und geht zunächst auf die Ausgangssituation und Zielvereinbarungen ein. Anschließend wird die Vorgehensweise bei der Modellierung beschrieben und auf die mathematischen Grundlagen sowie Herausforderungen im Umgang mit Protégé eingegangen.

Der zweite Teil beginnt mit einer Einführung in REST und Python sowie einem in Python verfassten Beispiel eines REST-Dienstes. Im Anschluss erfolgt eine Beschreibung, wie durch OSLC eine erleichterte Zusammenarbeit zwischen Anwendungen ermöglicht und mittels des Lyo-Frameworks umgesetzt werden kann.

Zum Abschluss erfolgt eine kritische Reflektion, inwiefern die verwendeten Methoden, Programme und Modelle zur Umsetzung der gewünschten Ziele geeignet sind.

## 2 Ausgangssituation

Am 13.11.2014 fand das Kickoff-Treffen zwischen den Bosch-Vertretern und der Projektgruppe statt. Innerhalb einer Präsentation wurde die momentane Situation im Unternehmen beschrieben. Bei der Durchführung eines Innovationsprozesses gibt es das Problem der Integration von Kundenanforderungen, Geschäftsmodell und Produktentwicklung über den gesamten Prozess hinweg. Zwischen den einzelnen Bereichen ist während des Prozesses bisher kein standardisiertes Kommunikationsverfahren vorhanden, für eine erfolgreiche Durchführung ist ein Datenaustausch allerdings zwingend erforderlich. Zusätzlich soll sich der Prozess an agilen Vorgehensweisen, wie sie oft in Start-up-Unternehmen vorhanden sind, orientieren. Es soll eine Produkt- beziehungsweise Technologieentwicklung ermöglicht werden, die sich verstärkt an den Anforderungen des Geschäftsmodells und den Kundenanforderungen orientiert und nicht nur technische Kriterien berücksichtigt.

## 3 Zielvereinbarungen

Das Ziel des Projektes ist es, die Kommunikation zwischen unterschiedlichen Abteilungen innerhalb eines Innovationsprozesses zu verbessern beziehungsweise zu ermöglichen. Einzelne Abteilungen eines Unternehmens erstellen im Verlauf des Innovationsprozesses Dokumente und sammeln Informationen, die für einen bestmöglichen Ablauf auch den anderen beteiligten Abteilungen zugänglich gemacht werden müssen. Dazu wird eine technische Umgebung benötigt, durch die dieser Austausch stattfinden kann. Innerhalb dieser muss es eine geeignete Repräsentation der Informationen beziehungsweise des Wissens des Unternehmens geben, die erweiterbar und intelligent ist. Zur Darstellung wird der Ablauf des Innovationsprozesses, der letztes Semester bereits innerhalb des Praxisprojektes Anwendungssysteme herausgearbeitet wurde, übernommen. Dieser wird in Abbildung 1 dargestellt. Ausgehend von den acht Dokumenten und acht Prozessen wird der Informationsfluss innerhalb des Innovationsprozesses modelliert und dabei der Fokus auf die Propagation der Daten gelegt.

### 3.1 Evaluierung alternativer Implementierung

Bevor mit der Modellierung begonnen werden kann, muss eine Entscheidung bezüglich einer geeigneten Repräsentationsform des Wissens getroffen werden. In einem gemeinsamen Brainstorming wurden vier mögliche Lösungsansätze diskutiert, die in der Tabelle 1 gegenübergestellt sind.

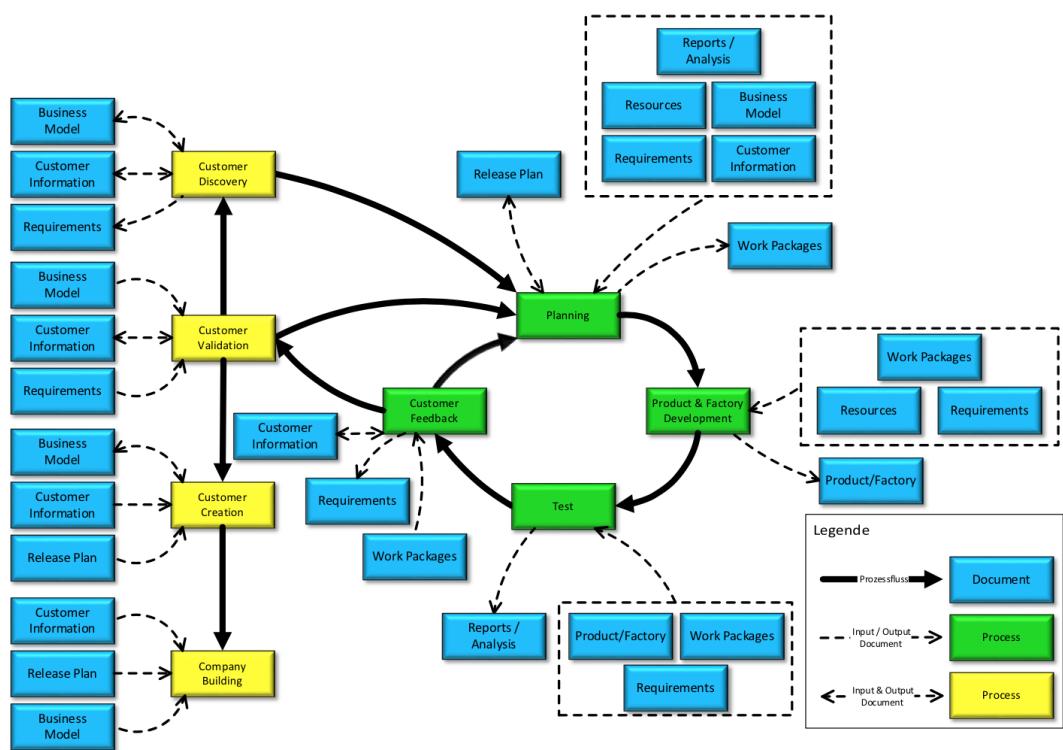


Abb. 1: Übersicht über Dokumente und Prozesse

Datenbank	neuronales Netzwerk
Beschreibung: System, das dazu dient, große Mengen von Daten zu speichern; Speicherung erfolgt strukturiert dauerhaft, widerspruchsfrei; benötigte Teilmenge der Daten kann durch Benutzer abgefragt und wie benötigt dargestellt werden	Beschreibung: Nachbildung der Verarbeitungsmuster des menschlichen Gehirns um Muster, Regeln und Beziehungen in großen Datenmengen zu erkennen; Erstellung verborgener Logikschicht, die die Eingaben verarbeitet und abhängig von Erfahrungen des Modells klassifiziert
Vorteile: für große Datenmengen geeignet, bedarfsgerechte Abfrage der Daten möglich	Vorteile: lernfähig anhand von Trainingsbeispielen, für komplexe Probleme geeignet
Nachteile: keine Speicherung der Bedeutung der Daten, keine Lernfähigkeit	Nachteile: Leistungsdefizite, wenn Training mit zu wenigen oder zu vielen Daten erfolgt; gelangen mit denselben Eingabedaten nicht immer zur selben Lösung
hybride Lösung	Ontologie
Beschreibung: die hybride Lösung stellt eine Mischform aus einer Datenbank und der Programmierung einer Anwendungssoftware dar, die es ermöglicht auf diese Daten zuzugreifen	Beschreibung: repräsentiert Wissen in einer bestimmten Domäne, in der relevante Begriffe durch ein Vokabular, Beziehungen durch Konzeptualisierung und Schlussfolgerungen durch Axiomatisierung dargestellt werden
Vorteile: keine Datenmigration notwendig; kann speziell auf Anforderungen eines Unternehmens zugeschnitten werden	Vorteile: durch Regeln über Zusammenhang der Daten und einer formalen Beschreibung dieser, lassen sich auch Informationen über die Bedeutung der Daten darstellen; lernfähig und erweiterbar
Nachteile: stellt keinen allgemein anwendbaren Lösungsansatz dar, sondern die Anwendungssoftware muss individuell angepasst werden; Erweiterung stellt sehr großen Aufwand dar	Nachteile: keine Möglichkeit von Berechnungen oder Operationen auf den Daten innerhalb der Ontologie

Tab. 1: Kurzübersicht Lernsysteme

### 3.2 Entscheidung für Ontologie

Der Ansatz der Ontologie wurde ausgewählt, da diese eine lernfähige Wissensrepräsentation darstellt, was in diesem Projekt ausgenutzt werden soll. Durch Inferenz- und Integritätsregeln werden Schlussfolgerungen gezogen und auf ihre Gültigkeit überprüft. So können auch logische Relationen zwischen Objekten dargestellt werden und die Ontologie ist erweiterbar und lernfähig. Die Ontologie stellt einen Wissensrahmen dar, der durch neue Einträge ergänzt werden kann, welche auf ihre Gültigkeit und logische Widersprüche geprüft werden.

### 3.3 Ziele für die Ontologie

Die entworfene Ontologie ist als Modell dafür zu sehen, welche Werkzeuge und Schnittstellen in einem Unternehmen benötigt werden, um Anwendungen unterschiedlicher Geschäftsbereiche zum Zweck der gemeinsamen Datennutzung und des Datenaustausches miteinander zu verbinden. Die Open Source-Software Protégé dient dem Erstellen und Validieren dieses Modells. Es ist nicht als Implementierungswerkzeug der tatsächlichen Datenverknüpfungen zu verstehen. Da nicht die Anschaffung und Evaluation einer Modellierungssoftware im Vordergrund stehen sollte sondern deren Verwendung, fiel nach einem kurzen Marktüberblick die Wahl aus folgenden Gründen auf Protégé:

- Frei verfügbar
- Vielzahl an Plugins
- Plugins ebenfalls frei verfügbar
- Dokumentationsbeispiele verwenden oft Protégé

## 4 Vorgehensweise bei der Modellierung

Die Erstellung einer Wissensrepräsentation erfolgt in Form einer Ontologie, da diese alle geforderten Eigenschaften besitzt. Das Vorgehen bei der Modellierung erfolgt iterativ und wird im folgenden Abschnitt detailliert beschrieben.

### 4.1 Umsetzung

Die Ergebnisse des Praxisprojektes Anwendungssysteme, welches im vorherigen Semester durchgeführt wurde, dienen als inhaltliche Basis der Ontologie. Jedes Projektmitglied hat sich einen Überblick über je einen der acht erarbeiteten Prozesse und eines der acht erarbeiteten Dokumente verschafft, wie sie in Abbildung 1

dargestellt sind. Um einen Gesamtüberblick zu erhalten, wurde eine *Matrix* erstellt, in der jedes Gruppenmitglied die von seinem Dokument benötigten Daten für jeden der acht Prozesse einträgt. Dazu wurden horizontal alle acht Dokumente und vertikal alle acht Prozesse eingetragen. Die Kästchen der Matrix beinhalten jeweils die entsprechenden Teile eines Dokuments, die in dem jeweiligen Prozess benötigt werden. Anschließend wurde nacheinander die Eintragung der Daten für die Dokumente und die Prozesse mit Hilfe des Ontologie-Editors Protégé vorgenommen. Bei der Eintragung des Dokuments **Business Model** wurde sich an dem Modell von Osterwalder und Pigneur [12] orientiert.

## 4.2 Protégé-Fenster

In Abbildung 2 ist ein Überblick des Protégé-Fensters zum Schnelleinstieg in das Programm zu sehen. Das Fenster unterteilt sich in die folgenden Bereiche, die je nach Auswahl des Reiters variieren. Zum Einstieg wird hier nur die Klassenansicht vorgestellt. Die Abweichungen in anderen Ansichten sind gering. Die Bereiche in der Abbildung sind im Text mit Zahlen in Klammern angegeben.

### 4.2.1 Class hierarchy-View

Die **Class hierarchy-View** (1) zeigt die Hierarchieebenen aller Klassen an. Direkt nach dem Start des Programmes werden nur die Klassen auf der höchsten Hierarchieebene dargestellt. Durch Klicken auf die dreieckigen Pfeilspitzen, die sich vor den Klassennamen befinden, kann jeweils die nächst tiefere Hierarchieebene angezeigt werden.

Die **Class hierarchy-View** dient der Navigation durch die vielen unterschiedlichen Klassen und ermöglicht einen guten Überblick über die Position der Klassen innerhalb der Ontologie.

### 4.2.2 Description-View

Die **Description-View** (2) gliedert sich in verschiedene Unterkategorien. In diesem Modell wird Gebrauch von **SubClass\_0f**, **Members** und **Disjoint With** gemacht. Die Unterkategorie **SubClass\_0f** dient zur Eintragung der Restriktionen. Hier können einer Klasse *Data Properties* und *Object Properties* direkt zugeordnet und durch die Operatoren **and**, **or**, **not** weiter eingeschränkt werden. Unter dem Punkt **Members** werden alle *Individuals*, die der jeweiligen Klassen angehören, automatisch aufgelistet. Durch die Eintragung einer Klasse unter **Disjoint With** kann festgelegt werden, zu welchen Klassen die jeweilige Klasse disjunkt ist.

#### 4.2.3 Annotation-View

Die **Annotation View** (3) dient dazu, einen besseren Überblick und erleichter-tes Verständnis der Ontologie zu erhalten. In Form eines `comment` vom Datentyp String wird hier eine kurze Erklärung der Bedeutung und Verwendung der Klasse gegeben. So können auch Personen, die sich vorher nicht intensiv mit dem Mo-dell der integrierten Produkt- und Geschäftsmodellentwicklung beschäftigt haben, die Bedeutung der einzelnen Klassen nachvollziehen. In diesem Modell sind alle Annotations in Englisch in Form von kurzen Stichpunktsätzen verfasst.

#### 4.2.4 Usage-View

Im **Usage View** (4) wird das Vorkommen der jeweiligen Klasse in der gesamten Ontologie dargestellt. Es werden alle *Individuals* der Klasse, die Erwähnung des Klassennamens in *Domains* beziehungsweise *Ranges* von *Object Properties* und *Data Properties*, die über `SubClass_Of` eingetragenen Restriktionen der Klasse selbst sowie das Vorkommen des Klassennamens in den Restriktionsangaben von anderen Klassen erwähnt.

#### 4.2.5 OntoGraf

Der **OntoGraf** (5) ermöglicht eine grafische Darstellung der Ontologie. Klassen werden hierbei als Kästchen mit gelbem Kreis, *Individuals* als Kästchen mit einer lilafarbenen Raute dargestellt. Die hierarchische Beziehung zwischen Klassen wird durch durchgezogene Pfeile dargestellt, die *Object Properties* durch gestrichelte Pfeile. *Data Properties* können durch `Node Tooltips`, die kleinen Notizzetteln äh-neln, dargestellt werden.

#### 4.2.6 SPARQL Query

Mithilfe der Abfragesprache SPARQL (6) können Klassen, *Data Properties*, *Object Properties* und *Individuals* abgefragt werden. Die Ergebnisse werden in Form einer Tabelle dargestellt.

### 4.3 Erstellung eines Ontologiegrundgerüsts

Nachdem die relevanten Begriffe der Rohdatensammlung aus der Matrix in die Ontologie eingetragen wurden, wird diese um redundante Einträge bereinigt und eine Hierarchie zur verbesserten Übersicht erstellt.

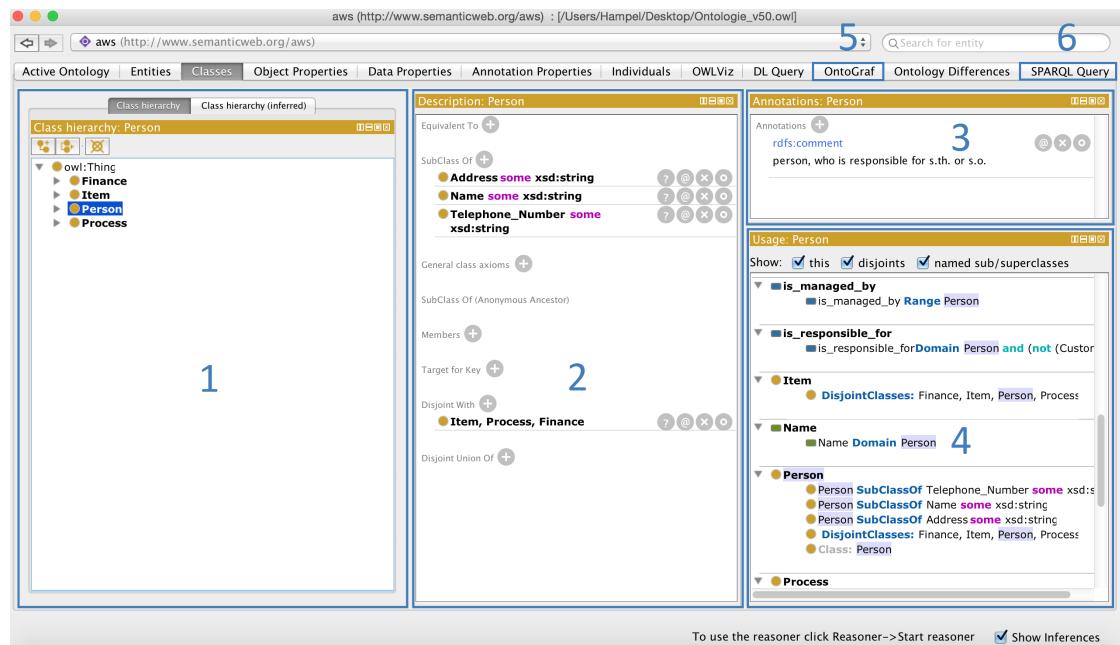


Abb. 2: Gesamtübersicht des Protégé-Fensters

Bei dem Entwurf der Hierarchie wurde sich für einen Top-Down-Ansatz entschieden und zuerst die vier Oberklassen **Finance**, **Item**, **Person** und **Process** entworfen, wie in Abbildung 3 zu sehen ist.

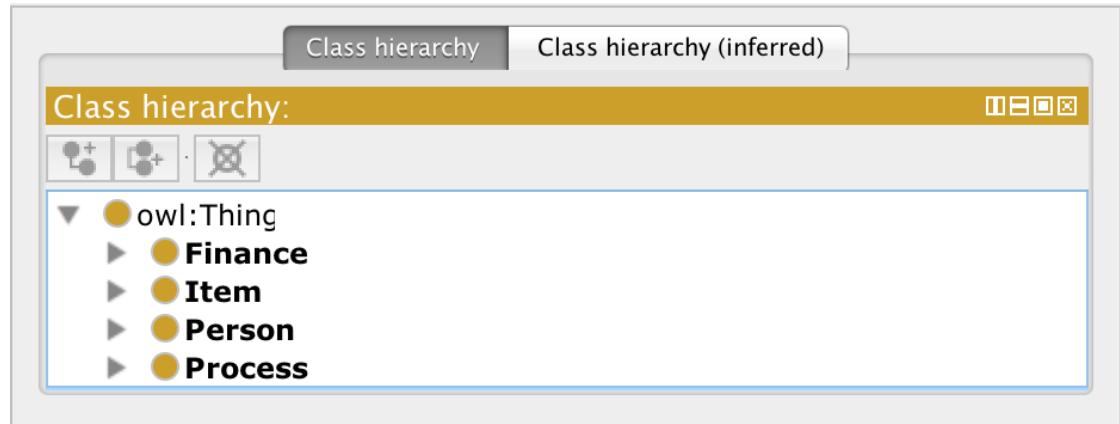


Abb. 3: Hierarchieübersicht der Ontologie

Es wurde beschlossen, die Klasse `Item`, wie in Abbildung 4 dargestellt, in die Unterklassen `intangible Item` und `tangible Item` zu unterteilen. Die Klasse `Document` wurde der Unterkategorie `intangible Item` untergeordnet, da sie als Informationsfluss, also immaterielles Gut, und nicht als Stück Papier in Form eines materiellen Gutes angesehen wird. Die Klasse `Prototype` wurde beabsichtigt nur der Klasse `Item` untergeordnet, da es virtuelle und materielle Prototypen gibt.

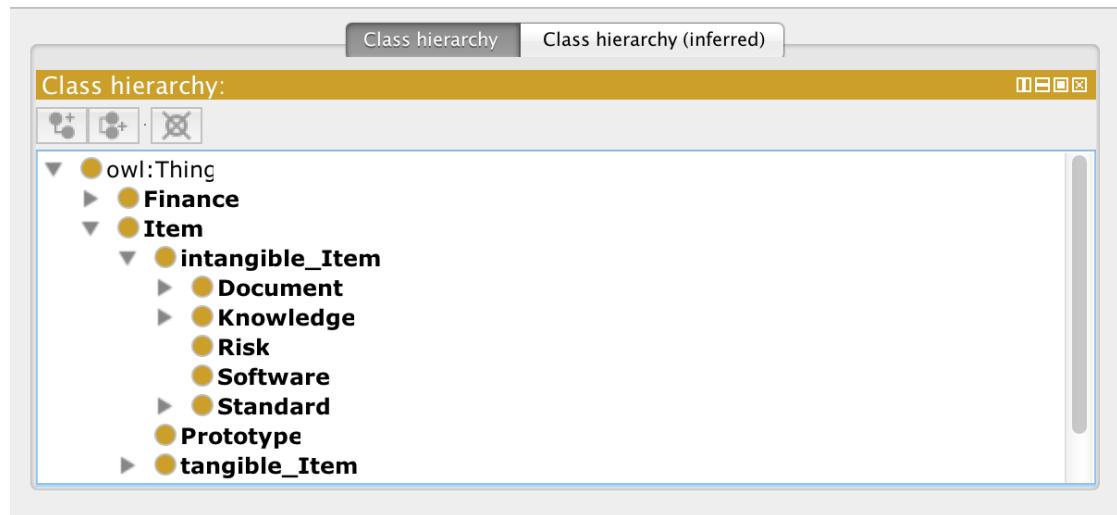


Abb. 4: Unterteilung der Klasse `Item` in `intangible Item`, `Prototype` und `tangible Item`

Anschließend wurden die restlichen bereits existierenden Klassen je genau einer dieser Oberklassen untergeordnet. Dadurch ist eine vereinfachte Navigation innerhalb der Klassenhierarchie möglich und die vier Oberklassen, sowie die zwei Unterklassen `intangible Item` und `tangible Item` stellen disjunkte Bereiche dar, was in Abbildung 5 und Abbildung 6 veranschaulicht wird.

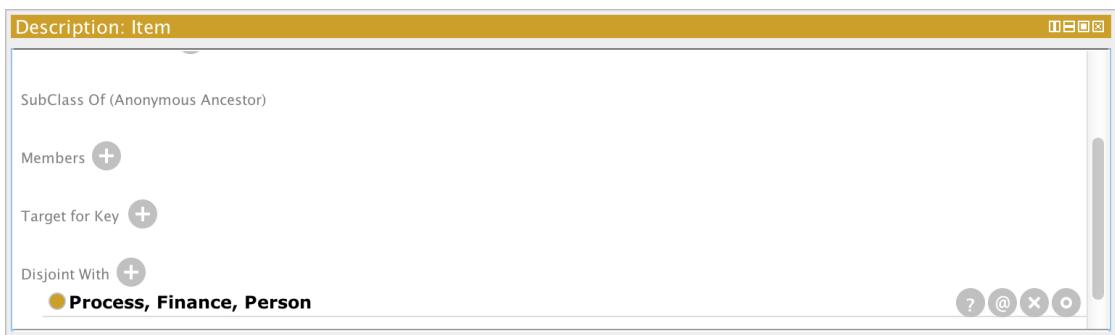


Abb. 5: die Klassen `Process`, `Finance` und `Person` sind disjunkt von der Klasse `Item`

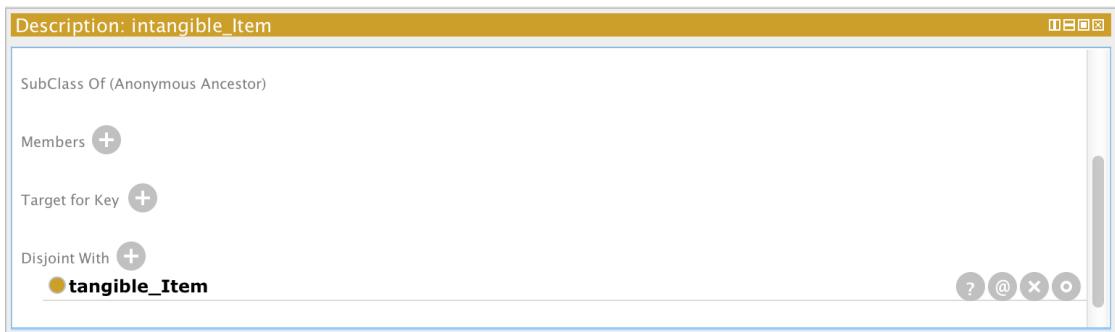


Abb. 6: die Klasse `tangible Item` ist zusätzlich disjunkt zur Klasse `intangible Item`

Die Erstellung von Unterklassen wurde zunächst auch zur Darstellung einer Teil-von-Beziehung genutzt. Zum Beispiel wurden die einzelnen Bestandteile eines Dokumentes als dessen Unterklassen eintragen. Das eigentliche Dokument stellt in diesem Fall die Oberklasse dar. Bei der erneuten Überarbeitung der Ontologie wurde die Hierarchieeinteilung so vorgenommen, dass die Verfeinerung in Unterklassen nur bei einer spezialisierten Beziehung möglich ist. Diese darf nur verwendet werden, wenn das Objekt der Unterkategorie stets auch als ein spezialisiertes Objekt der Oberklasse angesehen werden kann. Aus diesem Grund befinden sich nun zum Beispiel alle Dokumente und Teildokumente auf der gleichen Hierarchieebene, wie in Abbildung 7 zu sehen ist und eine Teil-von-Beziehung wird durch die Verbindung von zwei Klassen durch die *Object Property is\_Part\_of* vorgenommen.

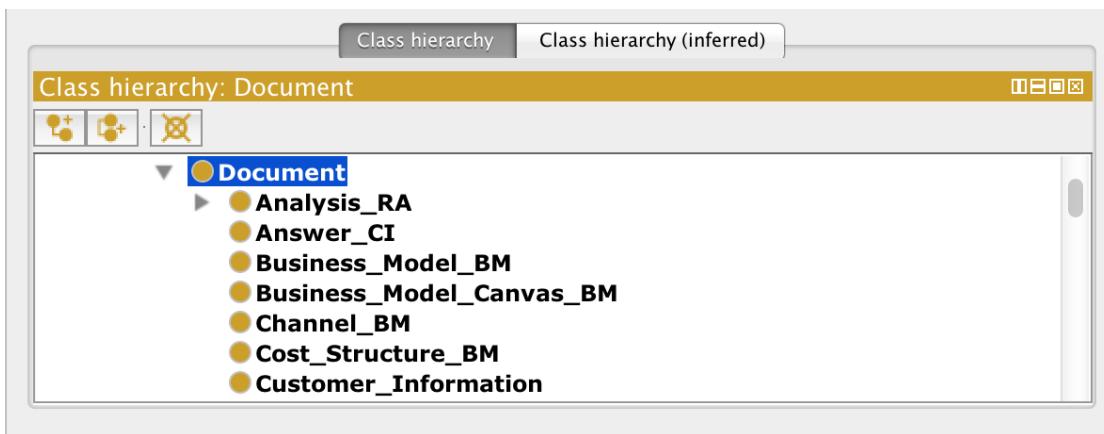


Abb. 7: Ausschnitt der Dokumentübersicht; **Channel\_BM**, der ein Teil des **Business\_Model\_Canvas\_BM** ist, befindet sich auf der gleichen Hierarchieebene wie das **Business\_Model\_Canvas\_BM**

Bei der Überprüfung der *Data Properties* wurde festgestellt, dass sehr viele redundante und ähnliche Attribute existieren. Es wurde sich daher darauf konzentriert einen allgemeingültigen Eigenschaftenkatalog zu entwickeln, der alle notwendigen Bezeichnungen enthält.

Bei den *Object Properties* lag der Fokus auf der Nachverfolgbarkeit der Beziehung, die invers, also in beide Richtungen, ermöglicht werden soll. Hierbei wurde sich auf die Vervollständigung der bereits existierenden Beziehung um ihre inversen Gegenspieler konzentriert.

#### 4.4 Dokumente und Prozesse verfeinern

Die in der Ontologie eingetragenen Dokumente werden verfeinert, indem ihnen *Data Properties* und *Object Properties* zugeordnet werden. Somit werden die bisher lediglich eingetragenen Begriffe in Beziehung zueinander gesetzt und Abhängigkeiten sichtbar gemacht.

Die Zuordnung der *Data Properties* und *Object Properties* zu den Klassen war in Protégé nicht intuitiv ersichtlich. Da in Dokumentationen und Tutorials oft reiner OWL- beziehungsweise RDF-Code zur Darstellung von Restriktionen verwendet wird, wurde Dr.-Ing. Stefan Fricke vom DAI-Labor an der Technischen Universität Berlin zu diesem Thema, speziell zur korrekten Eintragung von Restriktionen in Protégé befragt. Innerhalb von zwei persönlichen Gesprächen [6, 7] konnten alle programmspezifischen Fragen geklärt werden.

Es wurde die Entscheidung getroffen, die *Object Properties* und *Data Properties* über *SubClass Of* einzutragen, was in Abbildung 8 dargestellt ist.

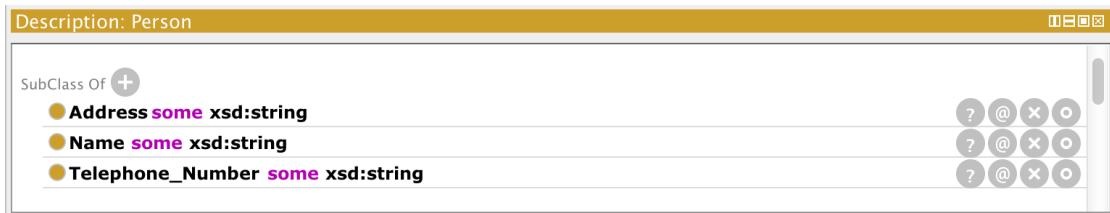


Abb. 8: Restriktionen werden über *SubClass Of* eingetragen

Aufgrund der multiplen Vererbung innerhalb der gesamten Ontologie wurden Beziehungen immer nur zu der höchstmöglichen Oberklasse festgelegt, wie zum Beispiel in Abbildung 9 bei der *Object Property changes*. Hier sind nur **Process** als *Domain* und **Document** als *Range* definiert und die Beziehung wurde nicht noch einmal explizit zwischen allen Unterklassen von **Process** und **Document** festgelegt.



Abb. 9: Definition von *Domain* und *Range*

Bei der Festlegung der *Domains* und *Ranges* für die *Object Properties* werden gleichzeitig die tiefstmöglichen Klassen beziehungsweise Unterklassen gewählt, so dass die Einschränkung so präzise wie möglich erfolgt. Wenn für das genaue Festlegen von einer *Domain* beziehungsweise *Range* mehr als eine Unterkategorie notwendig ist, werden diese mit **or**, wie in Abbildung 10, verbunden um Individuen aus beiden Unterklassen als *Domain* beziehungsweise *Range* zuzulassen.



Abb. 10: Verbindung von zwei Klassen in der *Domain* beziehungsweise *Range* durch **or**

Wenn genau eine Unterklasse ausgeschlossen werden soll, die nicht als *Domain* beziehungsweise *Range* festgelegt werden darf, wird dies durch **and not** in Verbindung mit der unerwünschten Klasse deklariert, wie in Abbildung 11 zu sehen ist.

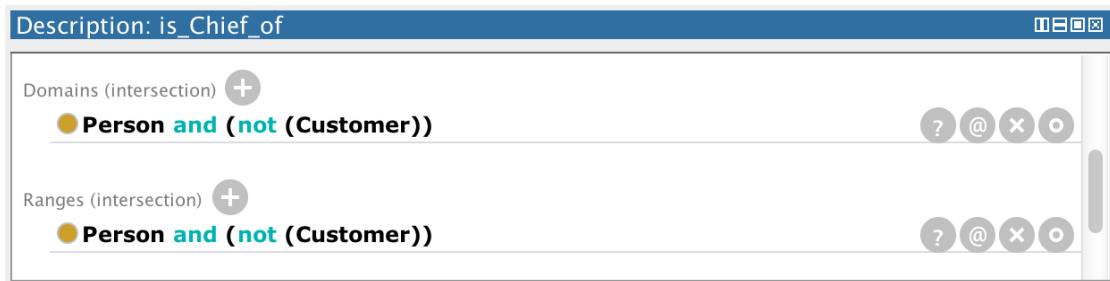


Abb. 11: Ausschluss einer Klasse aus *Domain* beziehungsweise *Range* durch **not**

So wurden beispielsweise dem Schaubild in Abbildung 1 entsprechend die durchgezogenen Pfeile als Vorgänger-Nachfolger-Beziehungen sowie die gestrichelten Pfeile als Input-Output-Beziehungen hinzugefügt, wie in Abbildung 12 zu sehen ist.

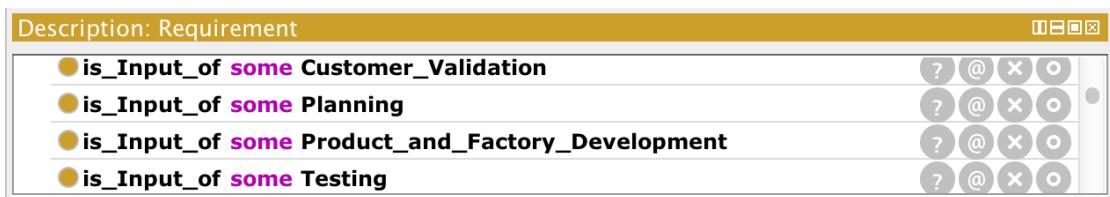


Abb. 12: Verknüpfung von *Requirement* mit Prozessen durch **is\_Input\_of**

Das Festlegen der *Data Properties* erfolgt in der *Domain* äquivalent zu den *Object Properties*. Da *Data Properties* allerdings nicht die Verbindung zwischen zwei Klassen, sondern die Zuordnung eines Attributes zu einer Klasse darstellen, wird bei der *Range* der gewünschte Datentyp des Attributes angegeben, was in Abbildung 13 gezeigt wird.

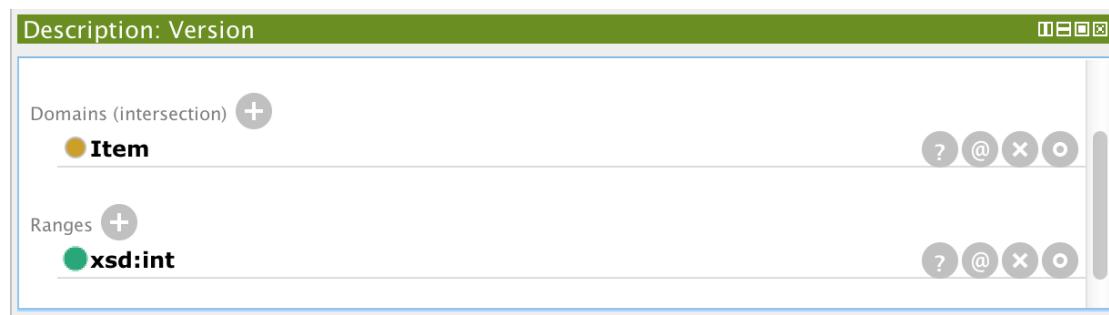


Abb. 13: *Domain* beziehungsweise *Range* einer *Data Property*

## 4.5 Reasoner und Cardinalities

Alle für die Ontologie relevanten Informationen sind eingetragen und miteinander in Beziehung gesetzt. Es werden disjunkte Beziehungen von Klassen zueinander festgelegt und *Domains* und *Ranges* von *Object Properties* definiert. Anschließend wird durch den *Reasoner* die Konsistenz der eingetragenen Restriktionen überprüft.

Zu jeder Beziehung wurde die inverse Beziehung explizit definiert, wie in Abbildung 14 für *is\_Part\_of* beispielhaft gezeigt ist.



Abb. 14: das Inverse der Beziehung von *is\_Part\_of* ist *consists\_of*

Die fehlerfreie Funktionsweise des *Reasoners* stellte am Anfang eine Herausforderung dar, da Hermit in Protégé 5.0 Beta sehr lange Berechnungszeiten und häufige Programmabstürze verursachte. Durch das Herunterladen der letzten stabilen

Version, Protégé 4.3, und der Verwendung des Reasoner-Plugin Pellet konnte eine fehlerfreie und schnelle Arbeitsweise des *Reasoners* sichergestellt werden. Nach Fertigstellung der Eintragungen wird der *Reasoner*, wie in Abbildung 15 dargestellt, gestartet.



Abb. 15: Start des *Reasoners*

Wenn die Ontologie widerspruchsfrei entworfen ist, werden die *Inferences*, also Rückschlüsse, wie in Abbildung 16 angezeigt, und der *Reasoner* ist aktiv.

Reasoner active  Show Inferences

Abb. 16: bei konsistenter Ontologie zeigt der *Reasoner* geschlussfolgerete Beziehungen und Eigenschaften an

Nachdem der *Reasoner* die Berechnung beendet hat, werden die geschlussfolgerten Beziehungen gelb unterlegt, wie in Abbildung 17 angezeigt.

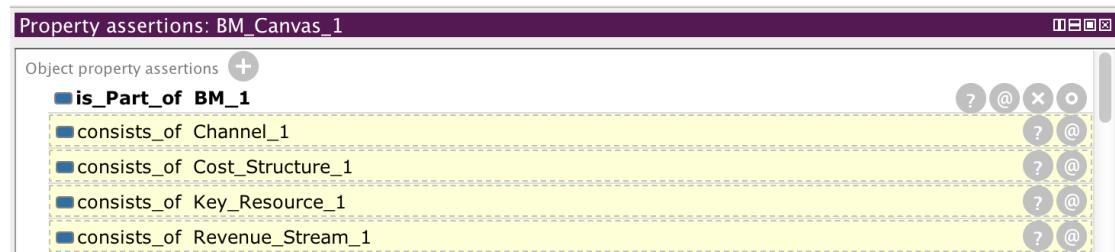


Abb. 17: der *Reasoner* schlussfolgert Beziehungen, die gelb unterlegt angezeigt werden

Theoretisch sind *Cardinalities* gut geeignet, um Falscheintragungen vorzubeugen oder bestimmte Sachverhalte besonders hervorzuheben. Die praktische Verwendung spezialisierter Kardinalitäten hat in Verbindung mit den Individuen des

Beispiele zu inkonsistenten Ontologien geführt, weshalb sie wieder grundsätzlich auf **some** gesetzt wurden. Alle Bilder dieses Abschnittes stammen deshalb von einer älteren Ontologieversion und haben gegebenenfalls leicht abweichende Bezeichner. Es folgen einige Beispiele, an denen die Kardinalitätensetzung demonstriert werden soll.

Grundsätzlich soll es von jedem Oberprozess nur ein *Individual* in einem Projekt geben, um Redundanzen zu vermeiden. Es soll einen **Planning**-Prozess, einen **Product\_and\_Factory\_Development**-Prozess und so weiter geben, aber beispielsweise niemals zwei **Planning**-Prozesse innerhalb eines Projektes. Deshalb soll jeder Unterprozess exakt einem Oberprozess zugeordnet sein, wobei diese auch keine, eine oder mehrere Instanzen eines Unterprozesses haben können.

Wenn ein einmal gewonnener Datensatz zu Kunden wiederverwendet werden soll, sollte, wie in Abbildung 18, **Customer\_Information** einige **is\_input\_of min 1**-Verknüpfungen mit mehreren Prozessen haben.

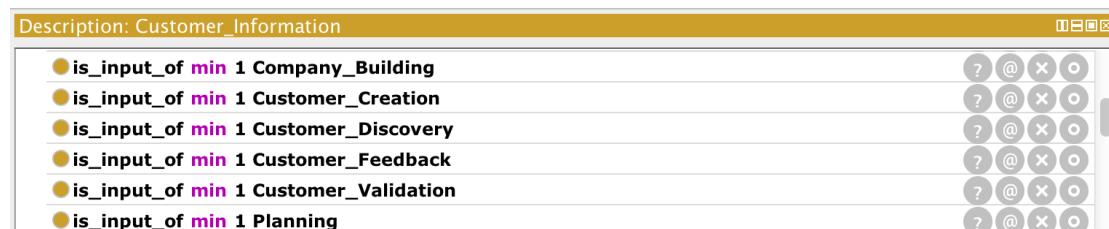


Abb. 18: **is\_input\_of**-Beziehung

Für eine Neukalkulation mit veränderten Werten soll kein neuer Prozess angestossen, sondern im gleichen neu bestimmt werden. Deshalb soll **Product\_and\_Factory\_Development** aus nur exakt einem **Estimate\_Delivery\_Date**- und **Estimate\_Development\_Costs**-Prozess bestehen.

Jede Hypothese wird durch einen eigenen **Evaluating\_Hypothesis**-Prozess bewertet, um zu verhindern, dass ähnliche Hypothesen zusammengefasst evaluiert werden, wodurch Fehler vermieden werden können.

Im Planungsprozess soll **min 1** Lösung im **Searching\_for\_Solution**-Prozess gefunden und deshalb auch **min 1** Lösung im **Evaluating\_Solution**-Prozess evaluiert werden.

Es existieren auch *Cardinalities*, die nicht näher spezifiziert werden können. Es gibt in der ersten Iteration des Planungszyklus keinen **Evaluating\_Planningprocess**-, **Suggesting\_Improvement**-Prozess oder Daten aus **Report\_and\_Analysis**, weshalb die Kardinalität an den entsprechenden *Object Properties* als **some** gesetzt werden soll.

## 4.6 Lernprozess

Da es sich bei einer Ontologie nicht um ein statisches Konstrukt handelt, ist es jederzeit erweiterbar. Wenn also im Innovationsprozess neue Unterprozesse entstehen oder neue Attribute in einem Dokument entwickelt werden, sollte dies auch in der Ontologie angepasst werden können.

Dabei muss immer darauf geachtet werden, dass keine *Classes*, *Data Properties* und *Object Properties* doppelt eingetragen werden, da sonst ein korrekter Informationsfluss nicht mehr gewährleistet werden kann.

### 4.6.1 Dokumente hinzufügen

Neue Hauptdokumente werden als Oberklasse eingefügt, die wiederum verschiedene Unterklassen, also verfeinerte Dokumente, haben können. Teildokumente, die mit einem Dokument in einer 'ist-Teil-von-Beziehung' stehen, dürfen, laut Definition der *SubClass*, nicht als Unterkategorie in die Ontologie eingetragen werden, sondern müssen zur Oberklassen werden.

Die Attribute der Dokumente bilden *Data Properties*. Mit Hilfe der *Object Properties* werden die Beziehungen zwischen einzelnen Dokumenten, aber auch zwischen Dokumenten und Prozessen beschrieben.

### 4.6.2 Prozesse hinzufügen

Das Eintragen eines Prozess verläuft ähnlich wie beim Dokument. Die Hauptprozesse werden als Oberklassen, Unterprozesse als Unterklassen eingetragen. Teilprozesse, die mit einem anderen Prozess in einer 'ist-Teil-von-Beziehung' stehen, dürfen, laut Definition der *SubClass*, nicht als Unterkategorie in die Ontologie eingetragen werden, sondern müssen zur Oberklassen werden.

Die Attribute der Prozesse werden als *Data Properties*, Beziehungen als *Object Properties* eingefügt.

### 4.6.3 Data Property hinzufügen

Beim Eintragen von neuen *Data Properties* sollte vorher überprüft werden, ob es sich wirklich um eine Eigenschaft handelt oder doch um eine Klasse. Ist Ersteres der Fall, sollte nach ähnlichen Kategorien gesucht werden und das neue Attribut gegebenenfalls als Unterkategorie eingetragen werden.

Als Letztes sollte ein sinnvoller Definitions- und Wertebereich eingetragen, sowie die neue *Data Property* den entsprechenden Klassen zugeordnet werden.

#### 4.6.4 Object Property hinzufügen

Wenn eine neue *Object Property* eingetragen wird, sollte vorher nach bereits vorhandenen Synonymen gesucht werden, damit es keine doppelten Beziehungen gibt. Der Vollständigkeit halber sollte immer die jeweilige Gegenrichtung der Beziehung angelegt werden, so werden in jeder Klasse die Beziehungen zu anderen Klassen sofort sichtbar.

Beispielsweise muss zur *Object Property* `is_Input_of` auch die Gegenrichtung `receives_Input_from` angelegt und diese als inverse Beziehung verbunden werden.

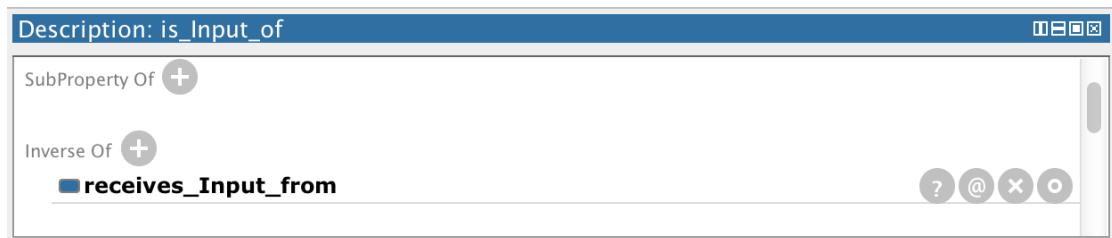


Abb. 19: *Object Property* mit Gegenrichtung

Wie bei den *Data Properties*, sollten hier auch geeignete Domains und Ranges gewählt werden. Beim Eintragen der Kardinalitäten, hat sich herausgestellt, dass man diese anfangs nicht zu eingeschränkt setzen sollte, da es sonst Probleme beim Erstellen der *Individuals* geben kann.

#### 4.6.5 Individuals erstellen

Wenn alle Prozesse, Dokumente, *Data Properties* und *Object Properties* eingetragen sind, ist die Grundstruktur der Organisation/des Unternehmens/des Innovationsprozesses abgebildet. Um mit realen Objekten zu arbeiten, müssen noch die *Individuals* angelegt werden, um die Ontologie mit Leben zu füllen. Diese sind die realen Instanzen der Prozesse und Dokumente und werden über die vorher angelegten *Object Properties* mit anderen *Individuals* verknüpft.

Während des Eintragens der *Individuals* kann es passieren, dass *Object Properties* fehlen, denn erst in diesem Schritt wird deutlich, welche Beziehungen notwendig sind. Die fehlenden Beziehungen müssen nachgetragen werden.

Am Ende sollte mit Hilfe des Reasoners die Konsistenz der Ontologie überprüft werden.

## 4.7 Modellstatistik

Nach Abschluss aller Modellierungsarbeiten beinhaltet das Modell die in Tabelle 2 gelisteten Informationsmengen:

Datentyp	Anzahl
Axioms	1978
Classes	129
Subclasses	353
Object Properties	37
SubObject Properties	8
InverseObject Properties	18
TransitiveObject Properties	4
Object Property Domains	20
Object Property Ranges	20
Data Properties	122
SubData Properties	104
Equivalent Data Properties	0
Disjoint Data Properties	2
Data Property Domains	31
Data Property Ranges	30
Individuals	126
Annotations	319

Tab. 2: Anzahl der Datentypen in der Ontologie

## 5 formale Grundlagen der Object Properties

*Object Properties* können verschiedene Eigenschaften haben. Im Folgenden werden die inverse und die transitive Relation näher betrachtet, da nur diese für das Beispiel relevant sind.

### 5.1 inverse Beziehung

Wenn es zu einer *Object Property* eine Gegenrichtung gibt, dann nennt man diese Relation inverse Beziehung.

#### 5.1.1 Definition

Für eine Relation  $R \subseteq A \times B$  ist die Umkehrrelation gemäß [11] definiert als

$$R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}.$$

#### 5.1.2 Beispiel

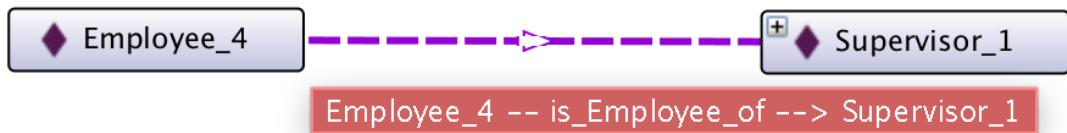


Abb. 20: OntoGraf: Relation `is_Employee_of` mit zwei Individuen

Wie in Abbildung 20 zu sehen, ist `Employee_4` dem `Supervisor_1` unterstellt. Zu dieser *Object Property* kann die Inverse gebildet werden.

Es sei  $A := Employee, B := Supervisor, a := Employee\_4, b := Supervisor\_1, R := is\_Employee\_of \Rightarrow is\_Employee\_of \subseteq Employee \times Supervisor$

Die Umkehrrelation lautet dann:

$$R^{-1} = \{(b, a) \in Supervisor \times Employee \mid (a, b) \in is\_Employee\_of\}$$

$$R^{-1} := is\_Chief\_of$$

Mit  $(a, b) \in R$  und der Existenz und Eindeutigkeit von  $R^{-1}$  folgt, dass  $(b, a) \in R^{-1}$ . Somit gilt:

$$\begin{aligned} b &:= \text{Supervisor\_1 und } a := \text{Employee\_4} \\ (b, a) &\quad \in \text{is\_Chief\_of} \\ \Rightarrow & \quad (\text{Supervisor\_1}, \text{Employee\_4}) \in \text{is\_Chief\_of} \\ \Rightarrow & \quad \text{Supervisor\_1 is\_Chief\_of Employee\_4}. \end{aligned}$$

### 5.1.3 Anwendung in Protégé

Diese inverse Beziehung kann auch in Protégé abgebildet werden. Sichtbar wird die Relation in der Liste der *Object Properties*, wo `Employee_4` ein Angestellter des `Supervisor_1` ist, siehe Abbildung 21. Im Folgenden wird die Ontologie einmal ohne und einmal mit *Reasoner* betrachtet.

Icon	Icon	Icon	Icon
?	@	X	O
?	@	X	O
?	@	X	O
?	@	X	O

Abb. 21: Properties des `Employee_4`

Die *Object Property* `is_Employee_of` wurde eingetragen, die Umkehrrelation aber nicht, da diese durch die Funktion des *Reasoners* geschlussfolgert werden soll.

#### ohne Reasoner:

Mit 6.4 SPARQL kann die Ontologie beispielsweise nach *Individuals* oder *Object Properties* durchsucht werden. Wenn also nach dem `Supervisor` gesucht werden soll, dem der `Employee_4` untergestellt ist, erscheint `Supervisor_1` als Ausgabe, siehe Quellcode in Listing 1 und Ausgabe in Tabelle 3.

Wird nach der Gegenrichtung der inversen Beziehung gefragt, wie in Listing 2, so bleibt Tabelle 4 leer. Dem System ist also der Chef von `Employee_4` nicht bekannt.

Listing 1: SPARQL-Abfrage: Relation `is_Employee_of`

```
PREFIX ont:<http://www.semanticweb.org/aws#>
SELECT ?employee ?supervisor
WHERE{
?class rdfs:subClassOf* ont:Person.
?supervisor a ?class .
ont:Employee_4 ont:is_Employee_of ?supervisor .
BIND( ont:Employee_4 AS ?employee) .
}
```

Employee	Supervisor
Employee_4	Supervisor_1

Tab. 3: Ergebnistabelle: Relation `is_Employee_of`

Listing 2: SPARQL-Abfrage: Umkehrrelation ohne *Reasoner*

```
PREFIX ont:<http://www.semanticweb.org/aws#>
SELECT ?supervisor ?employee
WHERE{
?class rdfs:subClassOf ont:Person.
?employee a ?class .
ont:Supervisor_1 ont:is_Chief_of ?employee .
BIND( ont:Supervisor_1 AS ?supervisor) .
```

Employee	Supervisor
/	/

Tab. 4: Ergebnistabelle: Umkehrrelation ohne *Reasoner*

### mit Reasoner:

Ist der *Reasoner* eingeschaltet, wird die Gegenrichtung der inversen Beziehung geschlussfolgert und gelb hinterlegt, vergleiche Abbildung 22.

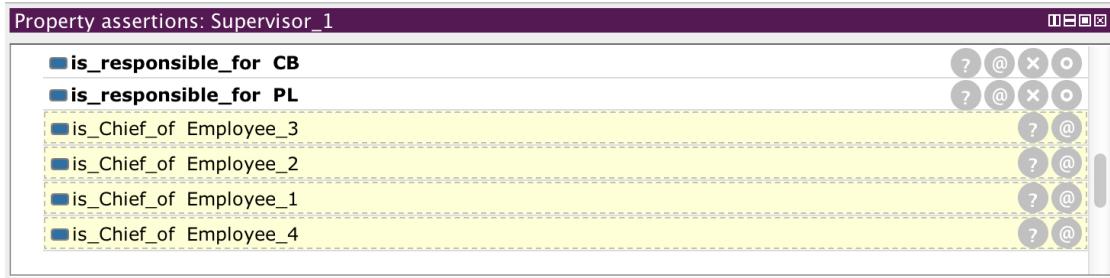


Abb. 22: geschlussfolgerete *Object Properties*

Nach einem 7.3 *Reimport* der Ontologie kann nun mit Listing 3 die Umkehrrelation abgefragt werden. Als Ergebnis wird Tabelle 5 zurückgegeben.

Listing 3: SPARQL-Abfrage: Umkehrrelation mit *Reasoner*

```
PREFIX ont:<http://www.semanticweb.org/aws#>
SELECT ?supervisor ?employee
WHERE{
?class rdfs:subClassOf ont:Person .
?employee a ?class .
ont:Supervisor_1 ont:is_Chief_of ?employee .
BIND( ont:Supervisor_1 AS ?supervisor) .}
```

Supervisor	Employee
Supervisor_1	Employee_1
Supervisor_1	Employee_2
Supervisor_1	Employee_3
Supervisor_1	Employee_4

Tab. 5: Ergebnistabelle: Umkehrrelation mit *Reasoner*

Im *OntoGraf* wird die Gegenrichtung wie in Abbildung 23 angezeigt. Da das Programm keine zwei Beschreibungen der *Object Properties* anzeigt, wurde die

Zweite in diesem Bild sowie in Abbildung 24 für eine bessere Anschaulichkeit manuell eingefügt und weicht daher vom tatsächlichen *OntoGraf*-Bild geringfügig ab.

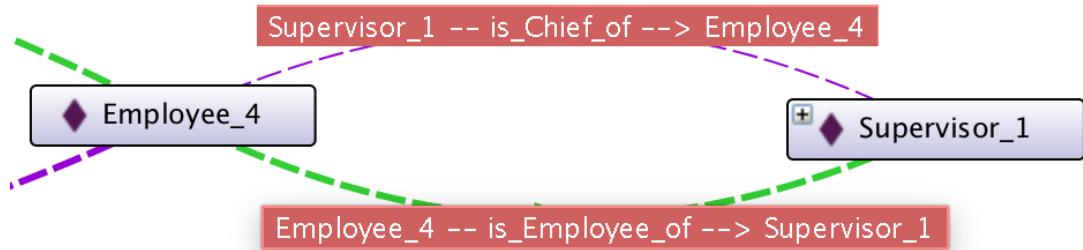


Abb. 23: *OntoGraf*: Relation `is_Employee_of` mit zwei Individuen

## 5.2 transitive Beziehung

Um bei einem Individuum eine Verkettung durch eine *Object Property* mit anderen Individuen darzustellen, wird die Eigenschaft der Transitivität benutzt.

### 5.2.1 Definition

Ist  $M$  eine Menge und  $R \subseteq M \times M$  eine zweistellige Relation auf  $M$ , dann heißt  $R$  gemäß [9] transitiv, wenn gilt:

$$\forall x, y, z \in M : xRy \wedge yRz \Rightarrow xRz$$

### 5.2.2 Beispiel

Für das Beispiel bedeutet das, dass dem `Chief_PD` sowohl der `Supervisor_1` als auch, durch die transitive *Object Property* `is_Employee_of`, der `Employee_4` unterstellt ist. Im *OntoGraf* ist es wie in Abbildung 24 dargestellt.

Sei  $R := \text{is\_Employee\_of}$  transitiv, dann gilt für  $M := \text{Person}$ ;  
 $\text{Employee\_4}, \text{Supervisor\_1}, \text{Chief\_PD} \in \text{Person}$ :  
 $\text{Employee\_4 is\_Employee\_of Supervisor\_1} \wedge$   
 $\text{Supervisor\_1 is\_Employee\_of Chief\_PD}$   
 $\Rightarrow \text{Employee\_4 is\_Employee\_of Chief\_PD}$

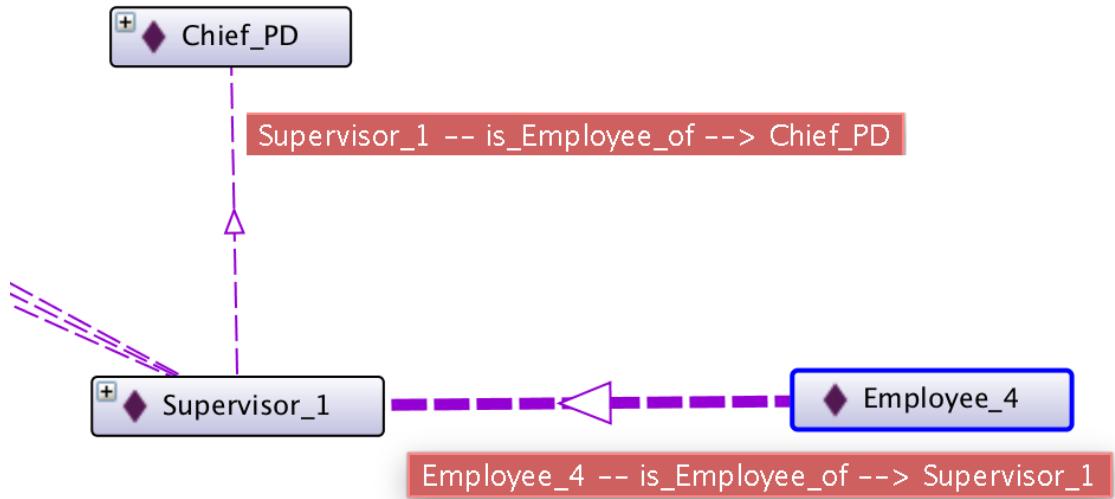


Abb. 24: *OntoGraf*: Relation `is_Employee_of` mit drei Individuen

### 5.2.3 Anwendung in Protégé

Die *Object Property* `is_Employee_of` wurde zwischen allen drei Individuen einge-tragen.

#### ohne Reasoner:

Ohne Schlussfolgerungen des *Reasoners* ist die Ausgabe der *SPARQL*-Abfrage die selbe wie im vorigen Beispiel.

#### mit Reasoner:

Bei eingeschaltetem *Reasoner* erscheint in Abbildung 25 die Relation zwischen **Employee\_4** und **Chief\_PD** gelb hinterlegt. Aus den Beziehungen von **Employee\_4** zu **Supervisor\_1** und von **Supervisor\_1** zu **Chief\_PD** erkennt der *Reasoner* also folgerichtig, dass alle drei Individuen zueinander in Beziehung stehen und **Employee\_4** somit über zwei Hierarchieebenen hinweg ein Angestellter von **Chief\_PD** ist. Auf diese Weise lassen sich zwischen gleichartigen Individuen Verbindungen aufbauen, die sich von einem zum anderen wie eine Kette fortsetzen. Typische Anwendungen sind die Modellierung von Abteilungen oder Personalhierarchien und Bauteile mit wiederverwendbaren und mehrfach auftretenden Baugruppen.

Abb. 25: geschlussfolgerete Beziehung

In der reimportierten Ontologie zeigt die *SPARQL*-Eingabe 4 alle geschlussfolgerten Relationen an, und es ist in Tabelle 6 zu sehen, dass `Employee_4` sowohl `Supervisor_1` als auch `Chief_PD` unterstellt ist.

Listing 4: SPARQL-Abfrage: transitive Beziehung mit *Reasoner*

```
PREFIX ont:<http://www.semanticweb.org/aw#>
SELECT ?employee ?supervisor
WHERE{
?class rdfs:subClassOf ont:Person .
?supervisor a ?class .
ont:Employee_4 ont:is_Employee_of ?supervisor .
BIND( ont:Employee_4 AS ?employee ) .
}
```

Employee	Supervisor
Employee_4	Supervisor_1
Employee_4	Chief_PD

Tab. 6: Ergebnistabelle: transitive Beziehung mit *Reasoner*

Im *OntoGraf* 26 ist die transitive Relation sichtbar.

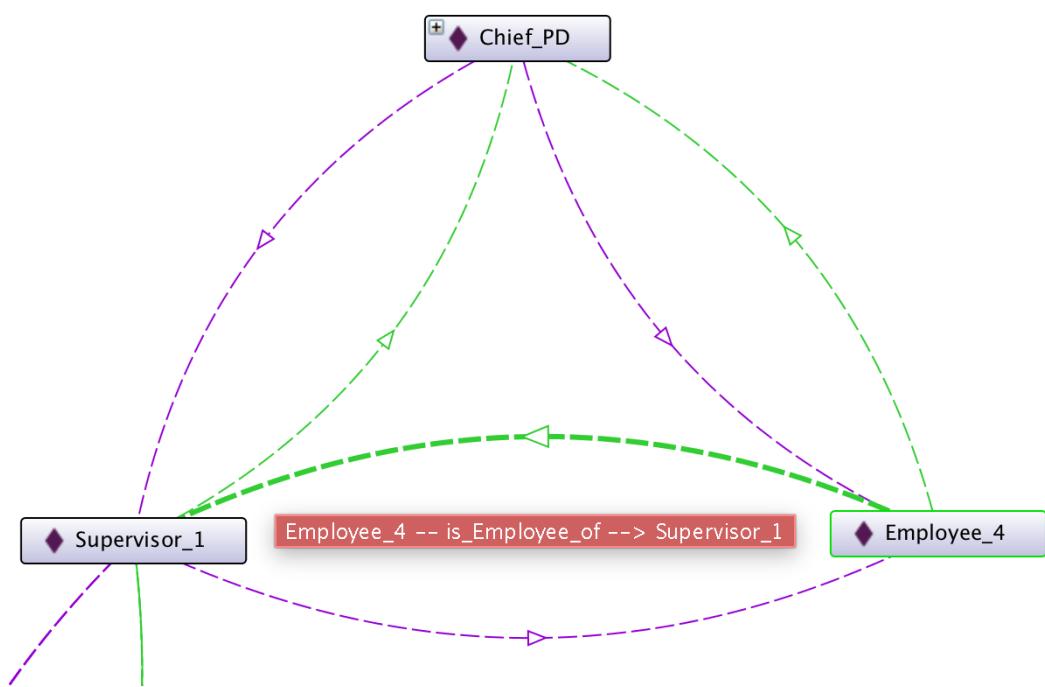


Abb. 26: OntoGraf: transitive Relation

## 6 Beispiel: Frosti, der Kühlschrank

Die Ontologie wurde vervollständigt und enthält alle wichtigen Prozesse, Klassen, *Object Properties* und *Data Properties*. Um die Ergebnisse anhand eines Beispiels zu demonstrieren, müssen in Protégé *Individuals* mit konkreten Werten erstellt werden. Den Abbildungen 27 und 28 ist zu entnehmen, dass diese untereinander mit den zugehörigen *Object Properties* in Relation gesetzt werden, um die Abhängigkeiten der *Individuals* von anderen zu verdeutlichen. Ihre Beziehungen werden von den *Object Properties* der Klassen abgeleitet. Mit Hilfe von *SPARQL*, einer graph-basierten Abfragesprache für RDF, können die Relationen abgefragt werden, um *Individuals* oder ihre Werte zu finden. Das Resource Description Framework (RDF)[10] ist ein Datenmodell zur Formulierung logischer Aussagen über einen Datensatz, welches auf einer definierten Semantik beruht. Diese logischen Aussagen werden dabei als Tripel, bestehend aus Subjekt, Prädikat und Objekt, modelliert.

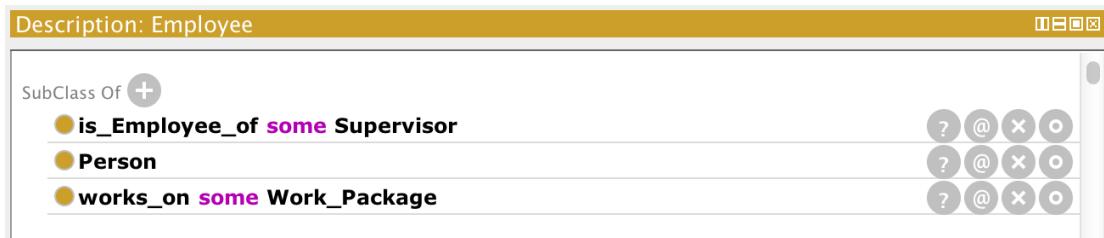


Abb. 27: Die Klassen der *Individuals* werden mit den zugehörigen *Object Properties* unter *SubClass \_ Of* Slot in Relation gesetzt.

### 6.1 Vorbereitung

Durch die Vervollständigung der Ontologie wurde deutlich, dass sie sehr komplex ist und nicht alle *Individuals* aller Klassen in einem Beispiel abgebildet werden können, ohne dass es unübersichtlich wird. Aus diesem Grund wird das Beispiel auf eine übersichtlichere Menge an Klassen reduziert. Diese Menge sollte eine signifikante Aussage über die ganze Ontologie geben. Das Beispiel bezieht sich auf die Grafik der Vorgängergruppe, welche in Abbildung 1 zu sehen ist. Bei dem Beispiel handelt es sich um die Produktion eines Kühlschranks.

### 6.2 Durchführung

Zu jedem Dokument und Prozess wurden *Individuals* erstellt. Nachdem diese mit ihren Eigenschaften erstellt wurden, konnten die Relationen von den Klassen ab-

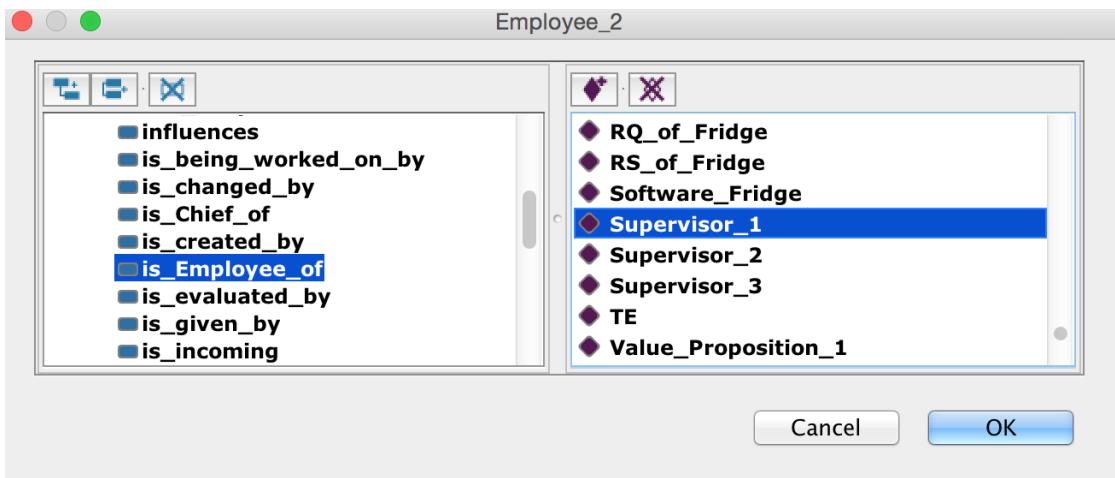


Abb. 28: Erstellung einer Relation zwischen den *Individuals* Employee\_2 und Supervisor\_1

gelesen werden und wurden mit Hilfe der *Object Properties* miteinander verknüpft. Um die *Individuals* erstellen zu können, wurde die dementsprechende Klasse ausgewählt, vergleiche Abbildung 29.

Wie in Abbildung 30 dargestellt, wurden mittels der *Data Property Assertions* die Eigenschaften der *Individuals* mit den jeweiligen Datentypen definiert und angelegt. Außerdem konnten durch die *Object Property Assertions* Beziehungen zwischen zwei *Individuals* hergestellt werden, siehe Abbildung 31.

Um die Ontologie dynamischer zu machen, wurden Personen, Rollen (*Chief*, *Employee*, *Supervisor* etc.) und Finanzen, wie beispielsweise der Bankkredit oder die Entwicklungskosten eingebunden, die ebenfalls Einfluss auf die Dokumente und die Prozesse nehmen, vergleiche Abbildungen 32 und 33. Die *Object Properties* und *Data Properties* der *Individuals* orientieren sich an den Werten, die in der zugehörigen Klasse vorgegeben wurden. Die Kardinalitäten wurden nachträglich geändert und alle auf *some* zurückgesetzt. Dies ist notwendig, damit man bestimmte *Object Properties* zu bestimmten *Classes* beziehungsweise *Individuals* ignorieren kann. Dies hat zur Folge, dass die Kardinalitäten der einzelnen *Object Properties* nicht der Realität entsprechen, jedoch für das Beispiel auch nicht notwendig sind.

### 6.3 Beispielablauf Produkt- und Geschäftsmodellentwicklung

Auf Basis dieser Verbindungen wird eine Beziehungskette zwischen den *Individuals* der Dokumente und Prozesse hergestellt, sodass ein beispielhafter Zyklus einer Produkt- und Geschäftsmodellentwicklung durchlaufen werden kann, wie in Abbildung 34 gezeigt wird. Das Beispiel dient dazu, die Abläufe in einem Unternehmen

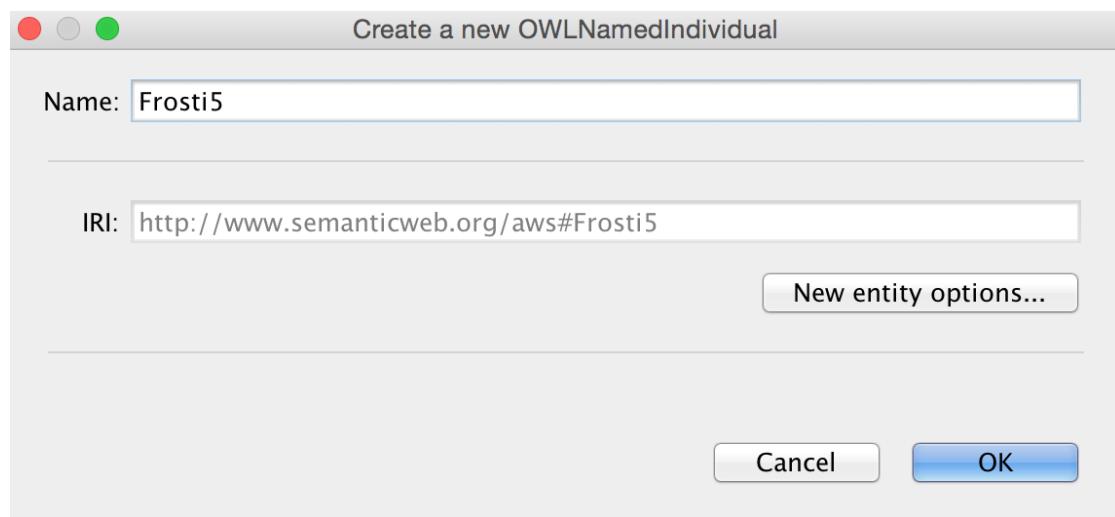


Abb. 29: Anlegen eines *Individuals*

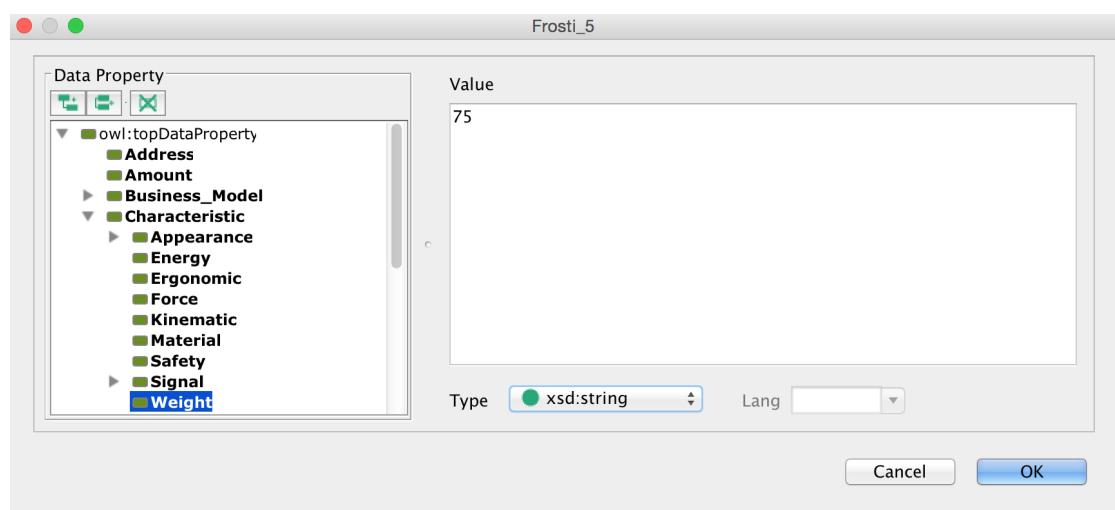


Abb. 30: Definieren und Anlegen von konkreten Eigenschaften

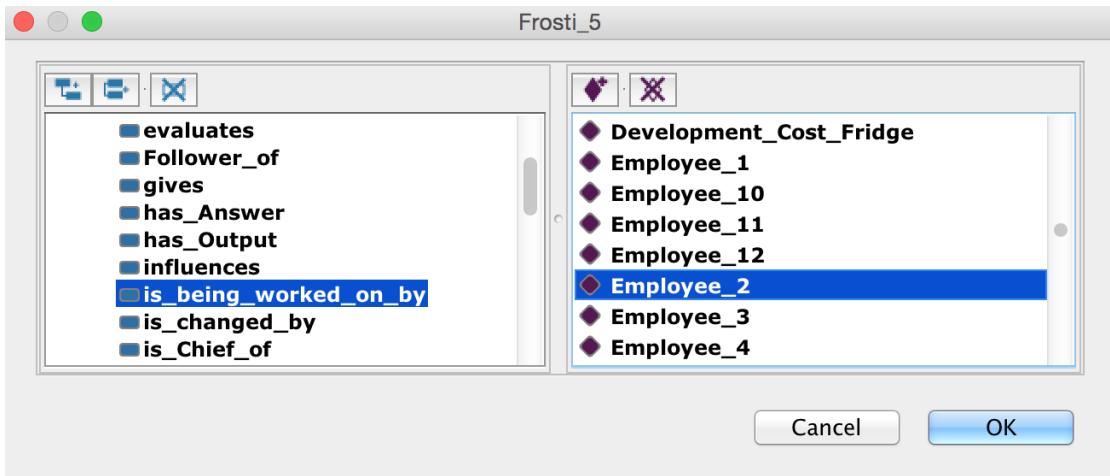


Abb. 31: Erstellung einer Relation zwischen den *Individuals* Frosti\_5 und Employee\_2

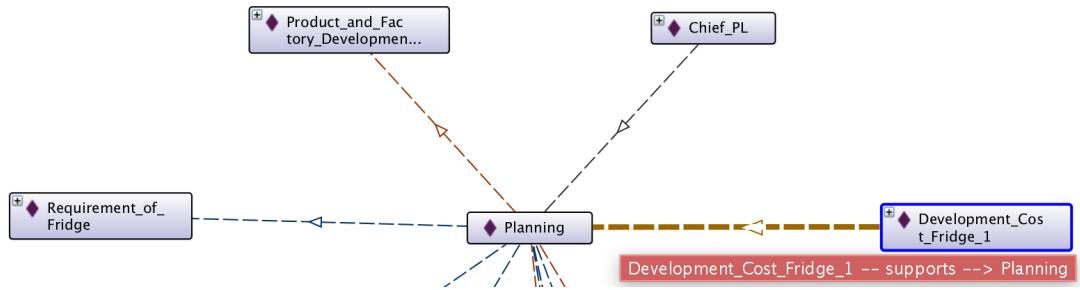


Abb. 32: Beziehung zwischen dem Planungsprozess und den Entwicklungskosten

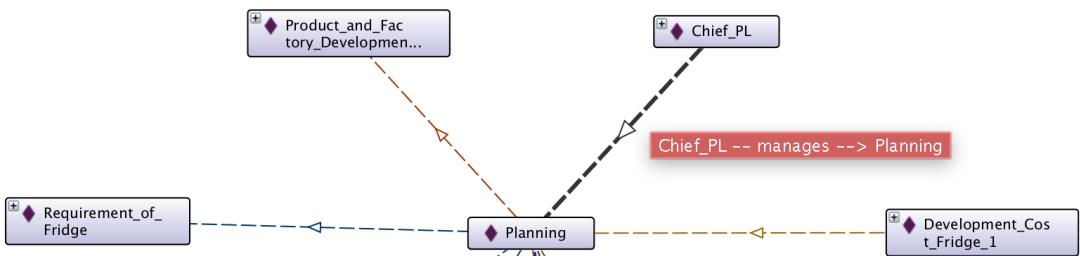


Abb. 33: Beziehung zwischen dem Planungsprozess und dem zuständigen Chief

vereinfacht darzustellen, um zu zeigen, dass auch bei einer Ausweitung auf einen vielfach komplexeren Unternehmensdatensatz die Integration und Propagation der Informationen einwandfrei funktioniert.

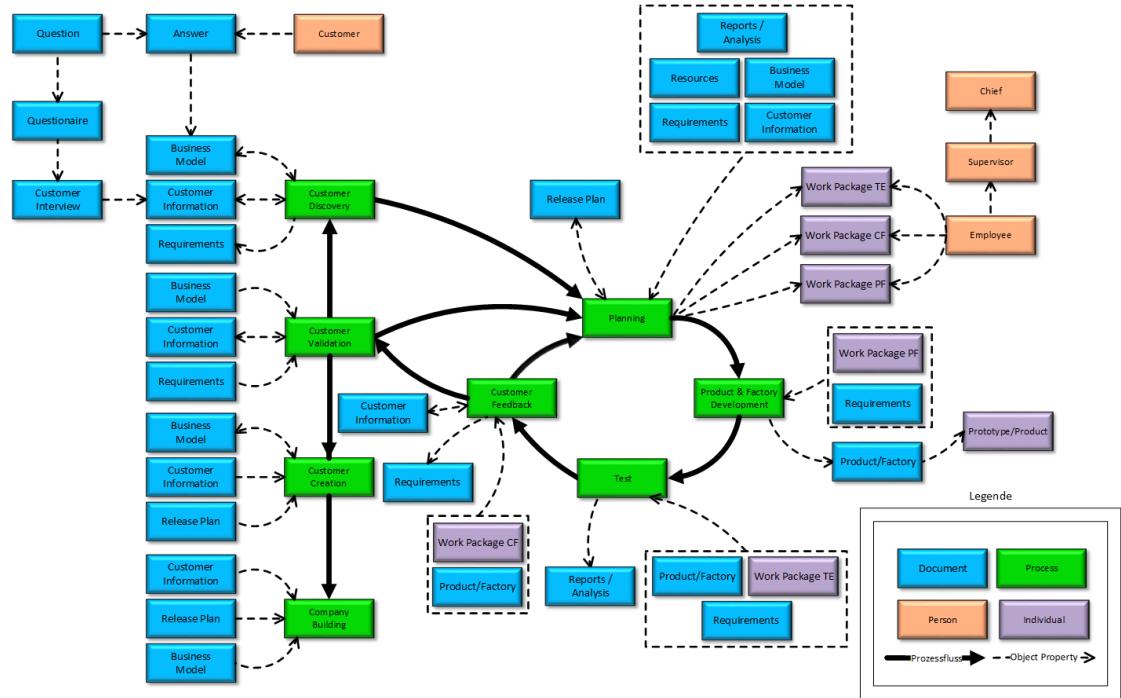


Abb. 34: grafische Darstellung des erweiterten Beispiels einer integrierten Produkt- und Geschäftsmodellentwicklung

Das Beispiel beginnt mit der Aufnahme der Kundenanforderungen. Es existiert ein **Questionnaire**, der aus mehreren **Question** besteht. Der **Questionnaire** ist Teil eines **Customer Interview**, welches wiederum einen Teil der **Customer Information** darstellt. Jede **Question** hat eine zugehörige **Answer**, die von einem bestimmten **Customer** gegeben wird. Die einzelnen **Answer** beziehen sich inhaltlich je auf einen der neun Teile des **Business Model** und stellen somit die in ein Geschäftsmodell umgewandelten Kundenanforderungen dar. Das **Business Model** beeinflusst die Prozesse **Customer Discovery**, **Planning**, **Customer Validation**, **Customer Creation** und **Company Building**. Im Anschluss an den Prozess der **Customer Discovery** folgt das **Planning**, welches als Output die **Workpackage** für die drei nachfolgenden Prozesse hat. Jedes **Workpackage** wird von einem **Employee** bearbeitet, der einem **Supervisor** unterstellt ist. Der **Supervisor** ist einem jeweiligen **Chief** untergeordnet, wodurch sich die gesamte Zuständigkeitsbeziehung für ein **Workpackage** nachvollziehen lässt. **Workpackage PF** bildet im Anschluss den

Input für den Prozess **Product & Factory Development**. Der Output dieses Prozesses ist das Dokument **Product/Factory** sowie der **Prototype** beziehungsweise das **Product**, sobald die eigentliche Herstellung beginnt. Der Prozess **Test** hat **Workpackage TE** als Input und **Report/Analysis** als Output. Als Input für den Anschlussprozess **Customer Feedback** dient das **Workpackage CF**, sowie das Dokument **Product/Factory**, da sich das Kundenfeedback immer auf ein bestimmtes Produkt/Prototypen bezieht. Als Ergebnisoutput des Prozesses entstehen Anpassungen durch die Kundenbewertungen im Dokument **Requirements**. Die Prozesse **Planning**, **Test** und **Customer Feedback** werden jeweils für den **Prototype** sowie anschließend für das **Product** durchlaufen. Nachdem das **Customer Feedback** für das **Product** gegeben wurde, folgen die Prozesse **Customer Validation**, **Customer Creation** und **Company Building**.

Diese Schritte stellen einen wichtigen Ausgangspunkt für die *SPARQL*-Abfrage dar, da es nun möglich ist, ganze Beziehungsketten mit ihren jeweiligen Verbindungen und Eigenschaften abzufragen. Beim Erlernen der *SPARQL*-Semantik und -Syntax wurde Herr Alexander Alexandrov zu Rate gezogen [2]. Anhand einer Beispiel-Abfrage wird die Funktionsweise von *SPARQL* erklärt.

## 6.4 SPARQL-Beispiel

Am Anfang jeder *SPARQL*-Abfrage werden Variablen deklariert. Variablen werden mit vorangestelltem Fragezeichen gekennzeichnet. Als Ergebnis der Anfrage im unteren Beispiel werden alle Variablenbelegungen für **class**, **name**, **zustaendig**, **amount**, und **investition** zurückgegeben, welche die definierten RDF-Tripel erfüllen. Weil das Ausschreiben der URIs die Leserlichkeit einer Abfrage mindert, können Präfixe verwendet werden. Hier steht ein **ont:** für: `<http://www.semanticweb.org/aws#>`.

Anmerkung zum *SPARQL*-Quellcode: Ein Prozentzeichen am Anfang einer Zeile dient dazu, eine Zeile auszukommentieren. Schaltet man diese Zeile frei, dann erweitert sich das Ergebnis der Abfrage. In dem folgenden *SPARQL*-Quellcode wird schrittweise jeweils eine Zeile hinten eingefügt, um ein besseres Verständnis zu verschaffen. So werden in vier Schritten nacheinander weiterführende Informationen zur Ermittlung eines Kredites herangezogen.

1. Abfrage: Alle Klassen und deren *Individuals* von der Klasse **Bank\_Lending** werden mit Listing 5 abgefragt, siehe Ergebnis in der Tabelle 7:

2. Abfrage: Listing 6 ergänzt die Ergebnistabelle um die Individuen, welche mit der Relation **manages** mit den *Individuals* von **Bank\_Lending** verbunden sind, die also für den jeweiligen Kredit zuständig sind. Tabelle 8 zeigt das Ergebnis dieser Hinzunahme:

3. Abfrage: Filterung mit Listing 7 der vorherigen Abfrage nach der Höhe des **Bank\_Lending**, die höher oder gleich 2000, aber niedriger als 3000 sein darf, das

Listing 5: SPARQL-Abfrage: *Individuals* der Klasse Bank\_Lending

```
SELECT ?class ?name ?zustaendig ?amount ?investition
WHERE {
?class rdfs:subClassOf* ont:Bank_Lending.
?name a ?class.
%?zustaendig ont:manages ?name.
%?name ont:Amount ?amount.
%FILTER ( ?amount < 3000 && ?amount >=2000) .
%?name ont:influences ?investition
}
```

Listing 6: SPARQL-Abfrage: Hinzunahme zuständiger *Individuals* der Kredite

```
SELECT ?class ?name ?zustaendig ?amount ?investition
WHERE {
?class rdfs:subClassOf* ont:Bank_Lending.
?name a ?class.
?zustaendig ont:manages ?name.
%?name ont:Amount ?amount.
%FILTER ( ?amount < 3000 && ?amount >=2000) .
%?name ont:influences ?investition
}
```

class	name
Bank_Lending	Credit_Deutsche_Bank
Bank_Lending	Credit_Commerzbank
Bank_Lending	Credit_Volksbank
Bank_Lending	Credit_Sparkasse

Tab. 7: Auflistung aller *Individuals* von der Klasse `Bank_Lending`

class	name	zustaendig
Bank_Lending	Credit_Deutsche_Bank	Chief_FO
Bank_Lending	Credit_Commerzbank	Chief_FO
Bank_Lending	Credit_Volksbank	Chief_FO
Bank_Lending	Credit_Sparkasse	Chief_FO

Tab. 8: Hinzunahme der Zuständigen des jeweiligen *Individuals*

Ergebnis ist der Tabelle 9 zu entnehmen:

Listing 7: SPARQL-Abfrage: Filterung

```

SELECT ?class ?name ?zustaendig ?amount ?investition
WHERE {
?class rdfs:subClassOf* ont:Bank_Lending .
?name a ?class .
?zustaendig ont:manages ?name .
?name ont:Amount ?amount .
FILTER ( ?amount < 3000 && ?amount >=2000) .
%?name ont:influences ?investition
}

```

4. Abfrage: Alle Prozesse, die durch die Relation `influences` mit dem `Bank_Lending` verbunden sind, werden durch Listing 8 abgefragt, vergleiche das Ergebnis in der Tabelle 10:

Mittels der Variablenbelegungen, welche Zeile für Zeile abgefragt werden können, kann eine Verkettung innerhalb verschiedener Klassen ermöglicht werden. Dies hilft zu einer strukturierter Ausgabe der benötigten Daten und stellt einen anschaulichen Zusammenhang für den Benutzer dar.

## 7 Herausforderungen in Protégé

Beim Aufbau des Datenmodells im Ontologie-Editor sind verschiedene konzeptuelle und programmbedingte Herausforderungen zu lösen, die hier kurz in Stich-

class	name	zustaendig	amount
Bank_Lending	Credit_Commerzbank	Chief_FO	2000

Tab. 9: Auflistung nur der *Individuals*, deren `amount` höher oder gleich 2000, aber weniger als 3000 hoch ist

Listing 8: SPARQL-Abfrage: Hinzunahme beeinflusster Prozesse

```

SELECT ?class ?name ?zustaendig ?amount ?investition
WHERE {
?class rdfs:subClassOf* ont:Bank_Lending .
?name a ?class .
?zustaendig ont:manages ?name .
?name ont:Amount ?amount .
FILTER ( ?amount < 3000 && ?amount >=2000) .
?name ont:influences ?investition
}

```

punkten zusammengefasst werden.

## 7.1 Modellierung der Objekteigenschaften

Bei der Modellierung in Protégé ist hauptsächlich die Aufgabe zu lösen, Lücken in der Dokumentation durch selbst entwickelte Heuristiken zu schließen und sich eine Menge von Best Practises zu erarbeiten:

- eine intuitive Herangehensweise beim Modellieren der Ontologie wird durch Protégé nicht gefördert, insbesondere ist die eindeutige Position der Restriktion nicht ersichtlich und erfolgt in diesem Beispiel über `SubClass_Of`
- verfügbare Plugins sind instabil oder erzeugen keine Ausgabe
- keine Rückmeldung des *Reasoners* für die Einhaltung von `min cardinality` und `exact cardinality`
- keine Dokumentation zu den Verknüpfungoperatoren `and`, `or`, `not`, die im freien *Class Expression Editor* verwendet werden können
- oft gibt es mehrere Möglichkeiten, eine Eigenschaft zu modellieren, zum Beispiel kann die genau-eine-Eigenschaftsdefinition über die `functional`-Definition der *Object Property Characteristics* oder durch die `exactly one`-Kardinalität bei der Definition einer Restriktion erfolgen

class	name	zustaendig	amount	investition
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	PL
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	Development_Budget_Fridge
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	PD
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	CC_for_Fridge
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	CV
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	TE
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	CD_for_Fridge
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	CB
Bank_Lending	Credit_Commerzbank	Chief_FO	2000	CF

Tab. 10: Hinzunahme aller Prozesse, die mit der Relation `influences` mit dem `Bank_Lending` verbunden sind

- nicht gewünschte Eigenschaften müssen explizit deklariert werden; wird zum Beispiel nicht gesagt, dass zwei Klassen disjunkt sind, können sie sich überschneiden

## 7.2 Reasoner für die Modellüberprüfung

Die *Reasoner* stellen den Benutzer vor die Aufgabe, wie allgemein die Vorgaben sein dürfen, aber wie streng sie sein müssen, um korrekte Ergebnisse und präzise Rückmeldungen im Fehlerfall zu erhalten:

- die *Reasoner* terminierten nicht, stürzten ab oder erzeugten Java-Exceptions und unzureichende Fehlermeldungen
- Reasoner-Plugin *Pellet* in Protégé 4.3 liefert bestmögliche Rückmeldung in kurzer Zeit
- Fehlermeldungen durch den *Reasoner* oft nicht eindeutig

Beispiel:

Die Klassen `Person`, `Item`, `Process` und `Finance` sind disjunkt. Die *Data Property Name* besitzt die *Domain Person*. Die Klasse `Channel_BM` stellt eine Unterklasse der Klasse `Document` dar. Wird angegeben, dass ein *Individual* von `Channel_BM` die *Data Property Name* besitzt, gibt der *Reasoner Pellet* folgende Fehlermeldung aus:



Abb. 35: Unterteilung der Klasse Item

Aus dieser Fehlermeldung ist nicht eindeutig ersichtlich, an welcher Stelle das Modell inkonsistent ist. Der Fehler kann sich auf alle Änderungen, die seit der letzten Überprüfung gemacht wurden, beziehen. In diesem Fall entsteht der Fehler durch ein *Individual* der Oberklasse *Document*, dem eine *Data Property* mit der *Range Person* zugeordnet wird, wodurch das *Individual* gezwungen wird, beiden Oberklassen anzugehören, um die geforderten Eigenschaften zu erfüllen. Da beide Oberklassen disjunkt sind, ist dies nicht möglich und das Modell ist inkonsistent. Da in der Regel mehrere Änderungen am Modell vorgenommen werden, ehe es neu überprüft wird, ist die Fehlersuche langwierig.

Die getesteten *Reasoner* funktionieren, so weit dies überprüft werden konnte, alle ähnlich. Das schließt den Umfang und die Aussagekraft ihrer Rückmeldungen an den Benutzer mit ein. Sie beschränken ihre Ausgabe darauf, Java-Exceptions zu generieren und maximal eine Fehlerstelle im Modell zu nennen, wie beispielsweise in Listing 9.

Listing 9: Beispielhafte Java-Exception

```
org.mindswap.pellet.exceptions.InconsistentOntologyException:  
    Cannot do reasoning with inconsistent ontologies!  
Reason for inconsistency: Individual http://www.semanticweb.  
    org/aws#Frosti is forced to belong to class http://www.  
    semanticweb.org/aws#intangible_Item and its complement  
at org.mindswap.pellet.KnowledgeBase.ensureConsistency(  
    KnowledgeBase.java:2076)
```

Mitunter erscheint kein Hinweisfenster zu prüfender Individuen, so dass die einzige Rückmeldung in Protégé selbst „is inconsistent“ am unteren rechten Fensterrand lautet. Für eine gezielte Fehlerdiagnose oder -suche ist das zu wenig. Die anschließend angebotene Möglichkeit, die Erklärungen für die Inkonsistenz aufzulisten, scheitert oft am nicht endenden Prozess „Computing explanations“. In einer frühen Version des Modells *Ontologie\_v20.owl* mit etwa 1200 Individuen war die Berechnung auf einem Core i7 mit 16GB RAM (davon 3,7GB zugewiesen an die JavaVM) nach wenigen Minuten bei sechs Treffern, nach über 19 Stunden jedoch nicht weiter fortgeschritten als „Computing explanations. Found 7“. Auch nach mehr als 28 Tagen war die Berechnung nicht fertig. Zu diesem Zeitpunkt wurde die Berechnung von Hand abgebrochen, weil ein deterministisches Ende nicht

mehr erwartet werden konnte. Protégé rechnete erkennbar wochenlang weiter, da während dieser Zeit weiterhin Logmeldungen wie Listing 10 in der Konsole erzeugt wurden.

Listing 10: Beispiel einer Logmeldung

```
OWLDataFactoryImpl.getInstance() WARNING: you should not use  
the implementation directly; this static method is here  
for backwards compatibility only
```

Ein Treffer für diese Warnmeldung<sup>1</sup> deutet darauf hin, dass es sich um einen Fehler in der Garbage Collection handelt, was die lange Laufzeit erklären mag. In Protégé 5.0 beta 15 war dieser Fehler nicht behoben, beta 16 und 17 erschienen erst wenige Tage vor Fertigstellung des Projekt und konnten nicht mehr berücksichtigt werden. An eine zeitnahe Überprüfung der Fehler im Modell ist damit nicht zu denken, was bei einem möglichen Praxiseinsatz dieses Werkzeugs berücksicht werden muss.

Andere *Reasoner* für Protégé wie FaCT++ und JFACT (basierend auf FaCT++) gaben ebenso wenige Rückmeldungen wie Pellet. Externe *Reasoner* arbeiten außerhalb von Protégé, sie basieren jedoch zumeist auf denselben Java-Bibliotheken<sup>2</sup>, oder es sind dieselben *Reasoner*, die bereits als Protégé-Plugin überprüft wurden, beispielsweise „Pellet: OWL 2 Reasoner for Java“<sup>3</sup> oder Hermit<sup>4</sup>.

### 7.3 Visualisierung und Abfrage mit OntoGraf und SPARQL

Der *OntoGraf* visualisiert das Modell aus Klassen, Individuen und Beziehungen, er kann jedoch nicht alle Attribute darstellen und nicht auf ungespeicherten, nur geschlussfolgerten Modellen arbeiten:

- kein Abfragen von geschlussfolgerten bzw. nicht-persistierten Beziehungen, das Modell muss inklusive der geschlussfolgerten Beziehungen gespeichert und anschließend neu importiert werden
- keine grafische Kennzeichnung des Pfades der SPARQL-Abfragen im *OntoGraf*
- Darstellung der *Object Properties* im *OntoGraf* nur einzeln und nicht dauerhaft möglich

<sup>1</sup><http://grepcode.com/file/repo1.maven.org/maven2/com.github.ansell.owlapi/owlapi-impl/3.4.3.3-ansell/uk/ac/manchester/cs/owl/owlapi/OWLDataFactoryImpl.java>

<sup>2</sup><http://owlapi.sourceforge.net>

<sup>3</sup><http://clarkparsia.com/pellet/>

<sup>4</sup><http://hermit-reasoner.com/java.html>

## 7.4 Kompatibilität des Dateiformats OWL

Aufgrund des standardisierten Dateiformats OWL/RDF sollte die Weiterarbeit mit einem anderen Editor ohne Einschränkungen möglich sein. Das funktionierte nur in wenigen Fällen. OWL-Dateien aus Protégé erzeugten beim Import in OntoStudio Java-Exceptions. Kleine Modelle aus der Anfangszeit ließen sich noch importieren, bei komplexeren Modellen ab v20 brach der Import jedoch ab. Über den Umweg des Exports als RDF statt OWL in Protégé konnte OntoStudio die Modelle zwar importieren, veränderte aber ungefragt die Struktur so deutlich, dass das zuvor exportierte Modell nicht mehr wiederzuerkennen war und letztlich neu hätte angelegt werden müssen. Bei einem Wechsel zu einem anderen Werkzeug muss deshalb damit gerechnet werden, das Modell noch einmal aufzubauen.

## 7.5 Stabilität des Modellierers und der Plugins

Protégé 4.3 als letzte Releaseversion gilt als stabil, wurde dem im Laufe dieses Projektes aber nur selten gerecht. Der Benutzer sieht sich häufig mit Programmabstürzen konfrontiert. Teilweise erscheinen diese in Form von Java-Exceptions und Stack-Auszügen in der Hintergrundkonsole, teilweise schließt sich das Programm kommentarlos und verwirft eventuell ungespeicherte Daten. In der aktuellen Beta-version 5.0 15 hat sich an diesem Zustand nichts geändert. Es funktionieren weniger Plugins als in 4.3, weil sie scheinbar neu angepasst werden müssen.

Da Protégé allem Anschein nach single-threaded arbeitet, werden die Möglichkeiten moderner Mehrprozessor- und Mehrkernsysteme nicht genutzt. Es blockiert während der Konsistenzprüfungsläufe, so dass es selbst auf schnellen Systemen häufig für 20 Minuten bis hin zu etlichen Stunden nicht ansprechbar ist. Hinzu kommt, dass nach jedem Laden einer Ontologie die Konsistenzprüfung erneut erfolgen muss, weil dieser Zustand nicht gespeichert wird. Zeitweise ist der Benutzer länger mit dem Warten auf Ergebnisse als mit dem Modellieren beschäftigt.

Bedenkt man die Selbstauskunft auf der Webseite <http://protege.stanford.edu>: „protégé is supported by a strong community of academic, government, and corporate users, who use protégé to build knowledge-based solutions in areas as diverse as biomedicine, e-commerce, and organizational modeling.“, so steht diese in deutlichem Widerspruch zur in diesem Projekt erfahrenen Qualität der Software, der Plugins und des Supports.

Der Vorteil von Open Source zeigt sich hier zugleich auch als Nachteil: im Mangel von Zuständigkeiten und Ansprechpartnern. Protégé gilt als das führende Werkzeug im Bereich der Ontologie-Modellierung. In der Praxis wurde es dem nicht immer gerecht. Das Programm selbst und mehrere Plugins verhielten sich instabil, es beendete sich im laufenden Betrieb mehrere Male ohne Vorwarnung, die *Reasoner* provozierten häufig Java-Exceptions oder nicht aussagekräftige Fehler.

lernmeldungen, Supportanfragen zum Programm selbst sowie zu fehlerhaften Webseiten und Plugins unterschiedlicher Autoren blieben unbeantwortet.

## 8 RESTful Web Services

REST ist die Abkürzung von Representational State Transfer [5]. Es ist kein Standard sondern eine Abstraktion der Struktur und des Verhaltens des World Wide Web.

Heutzutage erhöht sich die Geschwindigkeit, mit der sich das Internet ausbreitet. Dabei entsteht eine virtuelle Welt mit zahlreichen Informationen. Die Ressourcen des Internets können durch REST Web Services effektiver und effizienter angewendet werden. Weil sich diese auf eine Vielzahl von Servern leicht erweitern lassen und dem Benutzer erlauben, einen Webbrower als Client zu verwenden, sind die Anforderungen an die Software zu vereinfachen. REST stellt eine Alternative zu ähnlichen Verfahren wie SOAP und WSDL und dem verwandten Verfahren RPC dar.

Beim Surfen im Internet mit einem Brower müssen erst alle HTML, CSS, JSON und andere Daten heruntergeladen werden, dann ist die Website lesbar beziehungsweise sichtbar. Das traditionelle Vorgehen wird als ineffizient und unpraktisch angesehen, da sich die Informationen, über die man sich informieren möchte, in Latenz befinden. Durch REST lassen sich die im Internet verteilten Ressourcen besser abstrakt beschreiben und genauer positionieren. Ein unmittelbarer Austausch der erforderlichen Informationen zwischen REST-Server und REST-Client vereinfacht den obigen Prozess.

Eine REST-Schnittstelle ist eine potenzielle Sicherheitslücke, weshalb Server oder Datenbanken als primäres Angriffsziel betrachtet werden können. Der Angreifer könnte mit Administratorrechten die Daten ändern oder löschen beziehungsweise den Server überlasten, sodass es direkt zum Ausfall eines Dienstes führen kann.

### 8.1 Eigenschaften von REST

Folgende Eigenschaften muss ein REST-Dienst besitzen:

- Adressierbarkeit  
Jedes Element, das sich auf das Internet verteilt, kann durch den Uniform Resource Locator und Uniform Resource Identifier weltweit eindeutig adressiert werden.
- Unterschiedliche Repräsentationen  
Die Ressourcen lassen sich durch REST-Server in unterschiedlichen Formen

repräsentieren, deswegen können diese als verschiedene Datenformen exportiert werden, welche die Services anfordern. Beispiele sind HTML, JSON und XML.

- Zustandslosigkeit

Während der Client-Server Kommunikation soll der Kommunikationskontext auch verständlich sein, obwohl keine Zustände gespeichert werden. Das zustandlose Protokoll hilft dabei, die Anzahl der Verbindungen von HTTP-Requests erheblich zu senken.

- Operationen

REST benutzt die HTTP-Methoden GET, POST, PUT sowie DELETE, um Ressourcen zu manipulieren. Diese vier Operationen müssen orthogonal sein.

## 8.2 Python

Das unterstehende Liniendiagramm in Abbildung 36 gibt Auskunft über die Verbreitung der verschiedenen Programmiersprachen aus PYPL –PopularitY of Programming Language in 2014<sup>5</sup>. Daraus lässt sich ablesen, dass sich der Anteil von Python allmählich erhöht. Im Vergleich zu 2005 ist die Verbreitung von Python um etwa 300% gestiegen.

### 8.2.1 Vorteile

Python ist eine universelle und weit verbreitete Programmiersprache mit steigender Beliebtheit. Sie bietet als eine der ‘Interpreted’ Programmiersprachen eine kürzere Wartezeit bis zur Anzeige des Resultats und visuelle Ergebnisse direkt im Terminal an. Deswegen eignet sie sich gut für die Entwicklung von Rapid Prototype Models. Außerdem sind große Standardbibliotheken und zahlreiche Module verwendbar, damit sich Python in varierten Entwicklungsszenarien anpassen und sich der Zyklus der Entwicklung deutlich abkürzen lässt.

### 8.2.2 Nachteile

Python ist nicht zu verschlüsseln, so dass der originale Code für jeden lesbar ist. Mit Methoden wie Obfuscation kann man den originalen Code verschleiern. Da Python-Skripte im Interpreter laufen, geschieht die Abarbeitung relativ langsam. Außerdem benutzt Python nur allgemeine Fehlermeldungen, wie beispielweise ‘syntax error’ im Compiler, ohne die genaue Stelle des Fehlers zu benennen. Weil

---

<sup>5</sup> <http://pypl.github.io/PYPL.html>

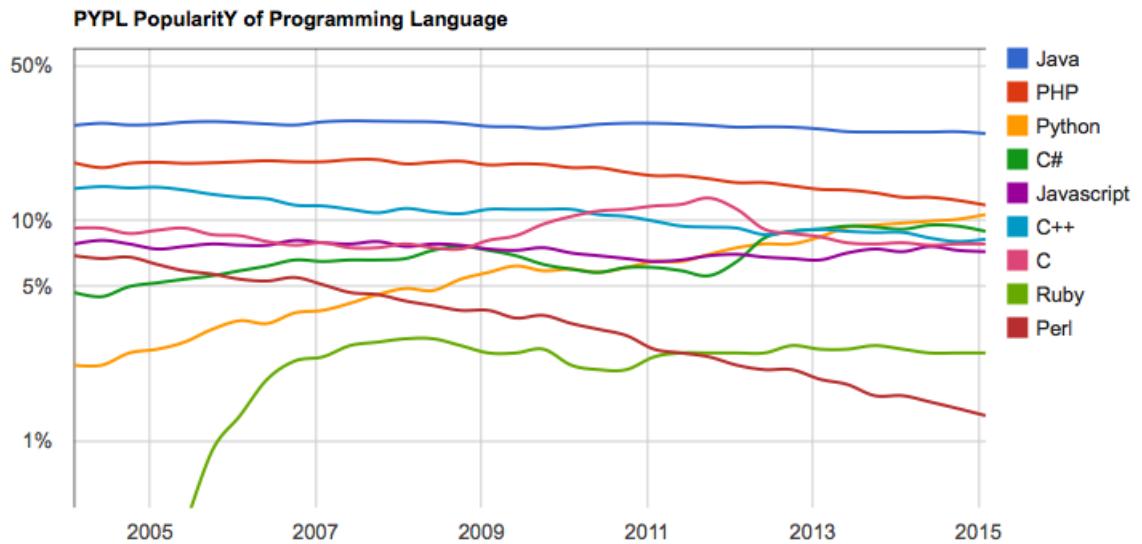


Abb. 36: Beliebtheit der Programmiersprachen

Python keine OS-Threads benutzt, unterstützt es weder Multi-Core noch Multi-Prozessoren. Der Code muss sich an die Syntax-Form halten, die in der vorhergehenden Code-Ebene definiert worden ist.

### 8.2.3 Code Obfuscation

Durch Code Obfuscation, auf Deutsch: Quelltextverschleierung, einen Begriff aus der Softwaretechnik, werden mit Absicht Veränderungen am Quellcode vorgenommen, damit der Code für Fremde schwer verständlich ist. Zahlreiche Methoden der Code Obfuscation sind verwendbar, zum Beispiel Anwendung von sinnlosen Wörtern als Variablennamen, Zeiger-Verwirrung (besonders in der Programmiersprache C) und vielfache Kodierung. Eine erfolgreiche Obfuscation des Codes hängt stark von der Fähigkeit des Programmentwicklers ab.

## 9 REST-Beispiel in Python

Das folgende Beispiel veranschaulicht die Grundfunktionen von REST.

## 9.1 Anforderungen an die Umgebung

Für die Lauffähigkeit des Python-Skripts sind die untenstehende Voraussetzungen zu erfüllen:

- Bugzilla 5.0+  
<https://bugzilla.mozilla.org> basiert auf Bugzilla, die REST-APIs sind identisch, und seit Bugzilla 5.0 steht natives REST-API [14] zur Verfügung, das derzeit auf <https://bugzilla.mozilla.org> läuft.
- Python 2.7/3.3/3.3+
- Betriebssysteme Windows XP, WIN 7, OS X 10.6+, Ubuntu 14+

## 9.2 Koordination mit Bugzilla-Server

Requests<sup>6</sup> ist eine HTTP-Bibliothek, die in Python geschrieben wurde. Außerdem wird das untenstehende Python-Skript Buxi dadurch unterstützt. Mithilfe von Requests sind REST-APIs leicht zu testen. Sie können beispielsweise als ein universeller REST-Client Get-Operation ausführen, sofern das Get-API der entsprechenden Website zur Verfügung steht.

Listing 11: Verbindung mit Bugzilla-Server

```
import requests
r = requests.get('https://bugzilla.mozilla.org/rest/login?
    login=username&password=password')
r.status_code
```

Nach Ausführung von Listing 11 ist ein Objekt mit dem Namen 'r' lesbar, zum Beispiel bedeutet der HTTP-Status-Code 200, dass die Verbindung erfolgreich hergestellt wurde. Die Variable 'r' ist frei gewählt und orientiert sich an den Python-Konventionen für die Namen von Variablen.

Listing 12: Suchen nach einer Bug-ID

```
url = 'https://bugzilla.mozilla.org/rest/bug'
u = url + '?id=1137137&include_fields=id&include_fields=
    summary'
search_results = requests.get(u)
search_results.text
```

Die Bug-ID <1137137> wird mit Listing 12 gesucht und als Suchergebnis werden ID und Summary als Textdatei exportiert.

---

<sup>6</sup><http://docs.python-requests.org/en/latest>

## 9.3 Python-Skript

Das Python-Skript heißt Buxi. Buxi ist der Name des Python Script Packages, der für interner Zwecke definiert worden ist. Es ist als REST-Service realisiert. Es wurde mit dem Funktionsumfang von GET, POST, PUT entwickelt. Es kann auf Bug-Informationen zugreifen, ein neues Bug-Ticket erstellen und Informationen eines bestehenden Bug-Tickets erneuern.

### 9.3.1 GET-Operation - Einen Bug einlesen

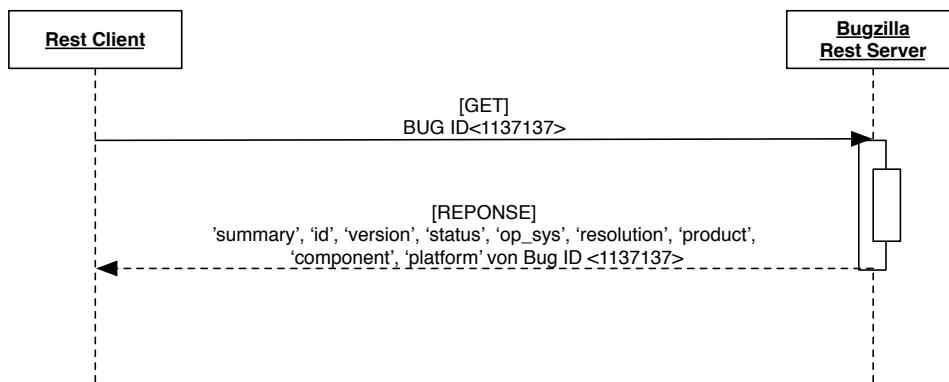


Abb. 37: GET-Prozessablauf im Sequenzdiagramm

Einen Bug zu suchen, ist ohne Login möglich, vergleiche Listing 13. Via REST-API wird eine Bug-ID gesucht. Wenn diese Bug-ID gefunden worden ist, werden alle Felder dieses Bugs, die vorher in 'DEFAULT\_SEARCH()' definiert worden sind, siehe Listing 14, an den Client zurückgesendet. Abbildung 37 beschreibt den Handshake zwischen REST-Client und Bugzilla-Server.

Listing 13: GET-Operation

```
def get(self, bug_number):
    bug = self.request('bug/%s' % bug_number,
                       params={"include_fields" :
                               self.DEFAULT_SEARCH}).json()
    return Bug(self, **bug['bugs'][0])
```

Als Ergebnis des Listings 15 wird die Summary des Bugs im Terminal angezeigt. Stattdessen kann auch auf andere Informationen zugegriffen werden, zum

Listing 14: DEFAULT\_SEARCH

```
class Buxi(object):  
  
    DEFAULT_SEARCH = ['version', 'id', 'summary', 'status',  
                      'op_sys', 'resolution', 'product',  
                      'component', 'platform']
```

Listing 15: Testbeispiel im Terminal für summary

```
import buxi  
bz = buxi.Buxi()  
bug = bz.get(1137137)  
bug.summary
```

Beispiel bug.version, bug.status, usw. Die Variablen 'bz' und 'bug' dürfen durch andere Möglichkeiten ersetzt werden.

**Spezieller Fall:** Sofern ein einzelner Kommentar benötigt wird, stehen die folgende Schritte zur Verfügung.

Listing 16: get\_comments-Funktion

```
def get_comments(self):  
    bug = unicode(self._bug['id'])  
    res = self._buxi.request('bug/%s/comment' % bug).json()  
    return [Comment(**comment) for comment  
            in res['bugs'][bug]['comments']]
```

Weil ein Bug nur ein Kommentarfeld besitzt, gibt es die Möglichkeit in diesem mehrere Kommentare zu hinterlassen, siehe Listing 16.

Nach Ausführung von Listing 17 wird der erste Kommentar im Terminal angezeigt. Der zweite wird mit `comments[1].text` abgerufen. Es gibt keinen Kommentar, wenn keine Antwort im Terminal zurückgesendet wurde.

Listing 17: Testbeispiel im Terminal für comments

```
import buxi
bugzilla = buxi.Buxi()
bug = bugzilla.get(1137137)
comments = bug.get_comments()
comments[0].text
```

### 9.3.2 POST-Operation - Ein Bug-Ticket erzeugen

Abbildung 38 erklärt den Ablauf beim Neuanlegen eines Bug-Tickets.

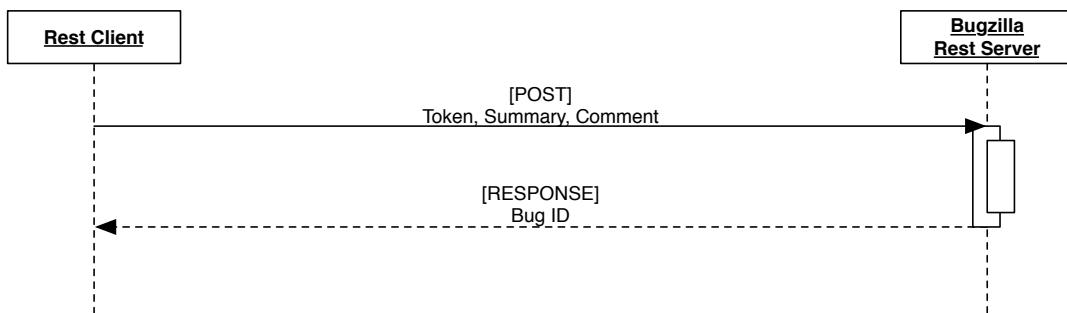


Abb. 38: POST-Prozessablauf im Sequenzdiagramm

Listing 18: Testbeispiel im Terminal für post

```
import buxi
bug = buxi.Bug()
bug.summary = ''this is a test demo''
bug.add_comment(''Do not be serious!'')
bz = buxi.Buxi("username", "password")
bz.put(bug)
bug.id
```

Werden Username beziehungsweise Passwort falsch eingegeben, wird eine Fehlermeldung verursacht und POST ist nicht erfolgreich. Mit Listing 18 werden ein Bug-Ticket mit Summary `this is a test demo` und ein Kommentar `Do not be serious!` gleichzeitig zu "bugzilla.mozilla.org" gesendet. Durch `bug.id` schickt der Bugzilla-Server die automatisch generierte Bug-ID zurück.

### 9.3.3 PUT-Information verändern

Abbildung 39 veranschaulicht, wie mittels der PUT-Methode die Statusinformationen an einem bestehenden Ticket verändert werden können.

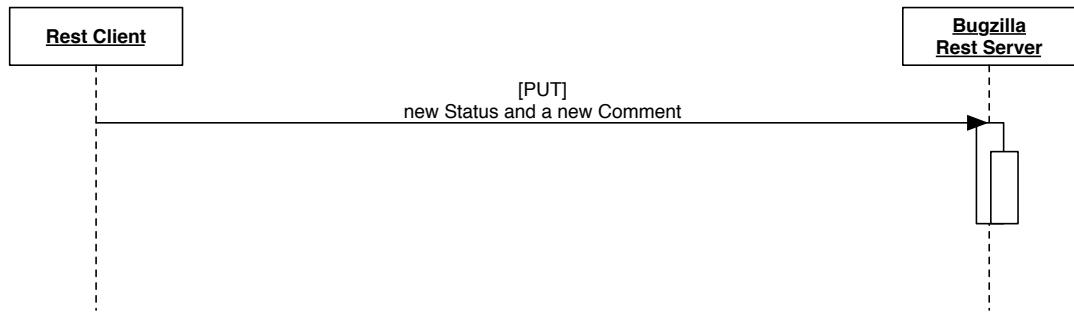


Abb. 39: PUT-Prozessablauf im Sequenzdiagramm

Damit ein Bug-Ticket verändert werden kann, muss die GET-Methode ausgeführt werden, um die notwendigen Informationen des Bugs zu erhalten, vergleiche Listing 19.

Listing 19: Testbeispiel im Terminal für put

```
import buxi
bz = buxi.Bug()
bug = bugzilla.get(1137137)
bug.add_comment(''this is my second test here'')
bugzilla = buxi.Buxi("username", "password")
bugzilla.put(bug)
```

Die weiteren Informationen sind auch veränderbar, zum Beispiel bug.OS, bug.resolution usw.

### 9.3.4 POST und PUT

Die POST- und PUT-Methoden werden in Listing 20 zusammen kombiniert, deswegen ist es möglich, ein Bug-Ticket damit herzustellen beziehungsweise zu erneuern. Um dies ausführen zu können, ist ein Login notwendig, damit ein Token dabei entsteht und gespeichert wird. Falls kein oder ein falscher Username beziehungsweise Passwort verwendet wird oder kein User existiert, führt das zu einem Fehler mit entsprechender Fehlermeldung.

Listing 20: POST- und PUT-Operationen

```
def put(self, bug):

    if not self.token:
        raise BuxiException('You have to login first')

    if not isinstance(bug, Bug):
        raise BuxiException('Bug Ticket Object missing',
                           )

    if not bug.id:
        result = self.request('bug', 'POST',
                              data=bug.to_dict()).json()
        if not result.has_key('error'):
            bug._bug['id'] = result['id']
            bug._buxi = self
        else:
            raise BuxiException(result['message'])
    else:
        self.session.post('%s/bug/%s' % (self.
                                         bugzilla_url,
                                         bug.id),
                          data=bug.to_dict())
```

## 10 Open Services for Lifecycle Collaboration

Softwareentwicklung ist ein komplexer Prozess, an dem eine Vielzahl von Systemen unterschiedlicher Hersteller beteiligt sind. Diese bringen individuelle, in der Regel zueinander inkompatible Schnittstellen mit sich. Open Services for Lifecycle Collaboration (OSLC<sup>7</sup>) ist ein Ansatz, um Anwendungen, die im Lebenszyklus von Software eingesetzt werden, die Zusammenarbeit zu erleichtern und diese Erleichterung den Entwicklern bereitzustellen.

---

<sup>7</sup><http://open-services.net>

## 10.1 Heterogene Strukturen in Organisationen

In Unternehmen und Organisationen mit gewachsenen Strukturen und komplexen Forschungs- und Entwicklungsprozessen existieren oftmals unterschiedliche und nicht oder begrenzt zueinander kompatible Programm- und Datensysteme. Insbesondere im Software-Entwicklungsprozess gibt es viele Lösungen mit aufwendigen Konfigurationen, um die Systeme miteinander zu koppeln. Im Entwicklungsprozess kommen unter anderem Werkzeuge für den Entwurf per UML, die Prototypenerstellung, die Umsetzung in Form von Quellcode, das Verwalten von Anforderungen (Requirements Engineering) und das Verifizieren der Umsetzung im Qualitätstest zum Einsatz. Zwischen diesen Anwendungen bestehen häufig nur einzelne Verbindungen, weil der Datenaustausch nicht normiert ist. Kommt es zudem zu Änderungen an einer dieser Schnittstellen, müssen zwangsläufig alle darauf basierenden Anwendungen geändert werden, um weiterhin auf die Daten zuzugreifen.

Von einzelnen Unternehmen gibt es integrierte Pakete, die oftmals sehr komplex sind und nicht benötigte Funktionen oder Abläufe mitbringen. Auf die Weise entsteht auch eine Abhängigkeit von einzelnen Anbietern, die ein Risiko in Hinblick auf die Langlebigkeit und Pflege solcher Systeme bergen. Gewünscht wird deshalb häufig eine individuelle Kopplung der vorhandenen Systeme, um nicht die komplette Software-Landschaft austauschen zu müssen und Umstellungskosten gering zu halten. Diese Kopplung muss zwischen je zwei Anwendungen erfolgen, die Daten miteinander austauschen sollen. Da sternförmige oder nahezu vollvermaschte Verbindungen zwischen allen Systemen angelegt werden müssen, steigt die Komplexität linear an, was den Aufwand für Direktverbindungen nur bei einer sehr geringen Anzahl eingesetzter Systeme erlaubt. Schon für fünf beteiligte Systeme mit bis zu 20 Schnittstellen untereinander ist ein solches Verfahren nicht mehr handhabbar, der Wartungsaufwand ist in den meisten Fällen zu hoch. Eine solche Umgebung erfordert ein universelles System zum Beherrschen dieses Aufwandes. Ein einheitlicher Industriestandard für diese Kopplung ist derzeit nicht in Sicht.

## 10.2 Ansatz: einheitliche Schnittstellen

Ein Ansatz zur Bewältigung dieser Aufgabe ist OSLC, das durch das Anbieten einheitlicher Schnittstellen unter Ausnutzung des REST-Paradigmas den Implementierungsaufwand verringern und so die Zusammenarbeit zwischen den Anwendungen erleichtern soll. Durch die Verwendung der REST-Technologien und mithin desselben Adressierungsschemas für Ressourcen im Netzwerk ergeben sich viele Gemeinsamkeiten mit semantischen Netzen (Semantic Web)<sup>8</sup> wie der hier verwendeten Ontologie. Somit bietet sich OSLC als Grundlage für eine prototypische Implementierung einer Verbindung zwischen REST-fähigen Systemen an,

---

<sup>8</sup><http://www.w3.org/2001/sw/wiki/ToolList>

um die Alltagstauglichkeit für die Anbindung unterschiedlicher Systeme und den entstehenden Aufwand zu untersuchen.

### 10.3 Reduzierter Implementierungsaufwand

OSLC schließt Lücken zwischen Systemen, die dieses Framework einsetzen, weil es den Implementierungsaufwand zum gemeinsamen Datenaustausch deutlich senkt. [13] Es etabliert ein gemeinsames Vokabular zum Verwalten von Informationen über Systemgrenzen hinweg. Das schafft ein gemeinsames Verständnis für ohnehin ähnliche oder gleiche Dinge und erleichtert die Zusammenarbeit für die beteiligten Personen.

### 10.4 Vorbild Internet

OSLC macht sich das Adressierungsschema von Informationen (Ressourcen) im Internet zunutze. Auf die Weise sind Ressourcen mit einem eindeutigen Pfad versehen und somit direkt ansprechbar. Durch Hinzunahme neuer Informationen ist dieses System beliebig erweiterbar, da die zugrundeliegenden Protokolle für diesen Zweck ausgelegt sind. Es skaliert gut, weil immer neue Ressourcen hinzukommen, ohne die alten einzuschränken, wie die permanent wachsende Anzahl an Hosts im Internet zeigt. Als verteiltes System ist es standortunabhängig: von jedem Punkt der Welt kann darauf zugegriffen werden. Jeder Zugriff ist demokratisch und gleichberechtigt: der Zugang steht allen Teilnehmern gleichermaßen offen.

### 10.5 Adressierbarkeit von Ressourcen

Im OSLC-Modell wird jede Ressource mit einem URL (Uniform Resource Locator) versehen. Das macht die Adressierung eindeutig, so dass jedes beteiligte Werkzeug direkt darauf zugreifen kann. Jede Ressource ist typenneutral, so dass unterschiedliche Materialien in einem einheitlichen System existieren können. Ressourcen können in unterschiedlichen Repräsentationen erfragt werden, beispielsweise kodiert als JSON, XML oder RDF. Dadurch stehen viele Anwendungen zur Wahl, um die Ressourcen zu adressieren und anzuzeigen. Webbrowser, Newsfeed-Reader und weitere Anwendungen sind etablierte Clients für solche Zwecke und stehen deshalb sofort als Anzeigeprogramme zur Verfügung. Da in der bevorzugten Repräsentation in XML für eine Ressource optionale Eigenschaften definiert sein dürfen, die von Anzeigern übergangen werden dürfen, wenn diese Angaben für sie irrelevant sind, ist das Basisformat sehr klein und fehlertolerant, umgekehrt aber leicht erweiterbar und änderbar. Insbesondere die hohe Fehlertoleranz ist ein gewichtiger Vorteil gegenüber herkömmlichen Dateiformaten, weil Daten aus einer Anwendung, die ei-

ne andere nicht interpretieren kann, diese nicht stören, jedoch um eigene Einträge erweitern.

In der Verwendung von RESTful Services für das Datenmodell ergibt sich ein großer Vorteil gegenüber anderen Datenorganisationsformen wie z.B. Datenbanksystemen, weil RESTful Services das bestehende Datenmodell um eigene Attribute erweitern können. Eine Datenbank wird dagegen von einem Datenbankmanagementsystem (DBMS) verwaltet, so dass sich andere Anwendungen wie RESTful Services hier nicht selbst anschließen können, um ihre Dienste anzubieten.

## 10.6 Services zur Verwaltung der Ressourcen

Ressourcen werden von Services verwaltet. Das OSLC-Modell verwendet hierfür RESTful Services, die jeweils kleine Einzelleistungen erbringen und so die bestehenden Daten um neue anreichern. Diese Architektur erübriggt Einzelanpassungen zwischen je zwei Werkzeugen und erlaubt das fallweise Arbeiten auf den gemeinsamen Daten. Auf diese Weise kann zum Beispiel ein fehlgeschlagener Build-Prozess in einem Werkzeug automatisch einen Fehlerbericht in einem anderen Werkzeug generieren, ohne dass hierfür eine neue Schnittstelle geschaffen oder das Dateiformat angepasst werden muss.

## 10.7 Gemeinsamer Namensraum

Aus der Herkunft der Daten ergeben sich unterschiedliche Domänen, in denen diese beheimatet sind. Eine Domäne kann zum Beispiel die Menge der Fehlerberichte zu einer Software sein (Quality Management) und eine andere die Menge der fehlgeschlagenen Kompiliervorgänge (Build-System) oder Änderungswünsche (Change Management), siehe Abbildung *40 Domänen des Lebenszyklusmanagements* auf Seite 56. Dank der Services können die Ressourcen aus einer Domäne auch in anderen Domänen sichtbar und editierbar gemacht werden. OSLC vereint sie in einem gemeinsamen Namensraum.

## 10.8 Basisoperationen CRUD

OSLC erlaubt das Ausführen der Basisoperationen beim Zugriff auf Datenhaltungssysteme, die so genannten CRUD-Operationen:

**Create - Retrieve - Update - Delete.**

Dieses sind die vier Grundoperationen zum Erzeugen, Empfangen, Aktualisieren und Löschen von Daten. Da die von OSLC verwendete REST-Architektur auf dem Hypertext Transfer Protocol (HTTP) basiert, entsprechen die CRUD-Operationen den HTTP-Operationen:

POST/PUT - GET - PUT - DELETE, siehe Vergleich in Tabelle 11.

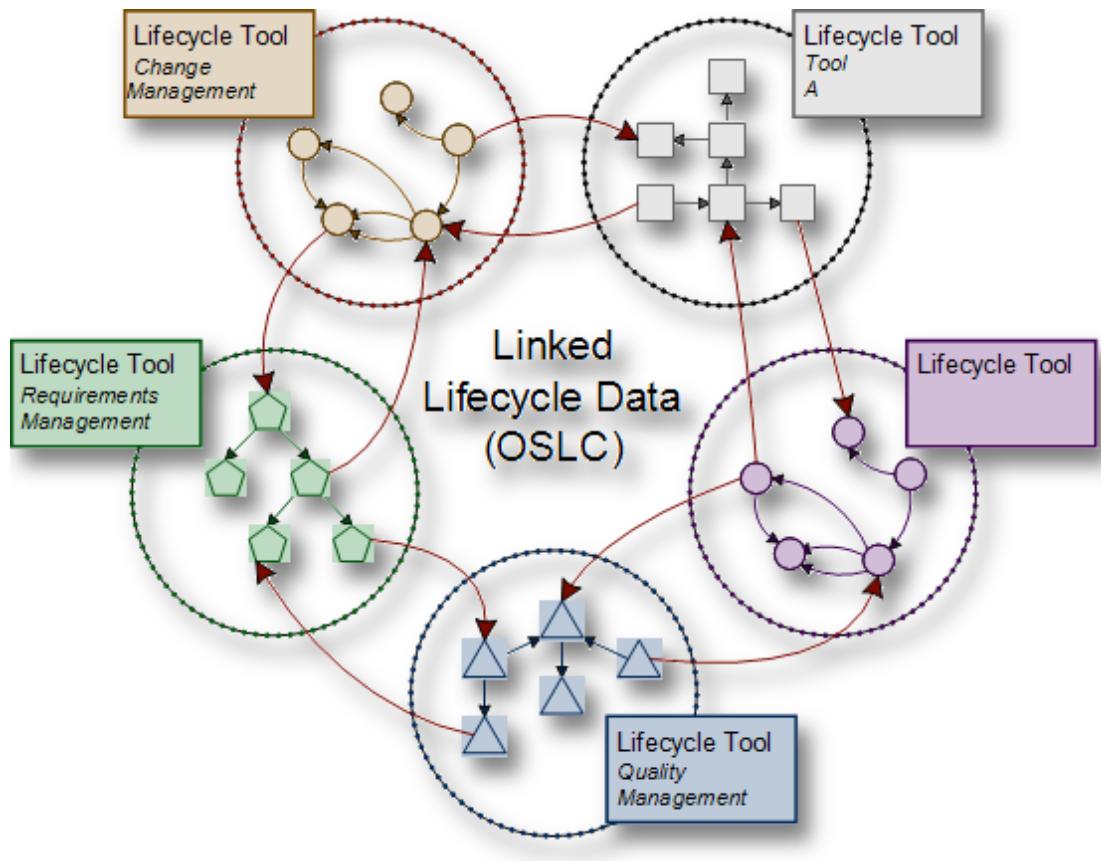


Abb. 40: Domänen des Lebenszyklusmanagements

Basisoperation	HTTP-Äquivalent
Create	POST/PUT
Retrieve	GET
Update	PUT
Delete	DELETE

Tab. 11: Basisoperationen CRUD

Über diese Operationen formuliert die lokale Webanwendung ihre Anfragen und nimmt Kontakt zur entfernt gelagerten Webanwendung auf.

Die Grundlage von HTTP ist ein wesentliches Argument für die Verwendung von OSLC, da etablierte Technologien und Programme weiterverwendet werden können. Dank des Austauschs in weit verbreiteten Datenformaten wie RDF/XML, JSON und Atom wird von Beginn an eine Vielzahl von Anwendungen unterstützt. Über diese Anwendungen hinaus erlaubt OSLC jedoch:

- ein selbst-kontrollierbares Vokabular zu erzeugen und verwalten
- ein Produkt oder System in seine Bestandteile zu zerlegen, da jede Ressource eine eindeutige Adresse erhält („product breakdown structure“)
- Dienste zum Verwalten des Wissens innerhalb der Domänen anzubieten
- Authentifizierung per HTTPAuth (empfohlen) oder OAuth

## 11 Umsetzung mittels Lyo/OSLC

Für die Anbindung unterschiedlicher Software an den gemeinsamen, mit Protégé modellierten, Namensraum wurde auf Vorschlag von Bosch das Lyo-<sup>9</sup>/OSLC - Framework eingesetzt. Lyo ist ein Framework für die Anwendungsentwicklung mit der integrierten Entwicklungsumgebung Eclipse. Das Framework erlaubt die Kommunikation per REST-Protokoll, so dass möglichst viele Anwendungen davon profitieren können. Eine Liste der unterstützten Systeme ist auf <http://open-services.net/software/> zu finden. Mit Hilfe der OSLC-Plattform wurde ein Prototyp zum Anbinden unterschiedlicher Server implementiert. Das Framework erlaubt das Anbinden von REST-Services, erfordert aber die Installation mehrerer, nicht konfliktfreier Pakete, weshalb zusätzlich ein minimalistischer Ansatz über ein Python-Skript verfolgt wurde. Das Python-Skript bietet den Vorteil, sich in bestehende Systeme relativ leicht einbinden oder gar autark als REST-Client auf der

---

<sup>9</sup>Eclipse Lyo - <http://eclipse.org/lyo/>

Kommandozeile starten zu lassen. Zu diesem Zweck kommt es ohne grafische Benutzeroberfläche aus. Diese kann am besten das jeweilige Programm bereitstellen, das das Python-Skript aufruft. Dagegen erlaubt das Lyo-Framework einen universellen Ansatz mit einer größeren Infrastruktur zu dem Preis, dass wesentlich mehr Vorarbeit erforderlich ist. Diese und der prototypische Aufbau wurden im Rahmen dieses Projektes geleistet.

Hier werden die Ergebnisse dieses Aufbaus behandelt, Details zur Umsetzung können dem Kapitel 17 *Implementierungshandbuch* auf Seite 77 entnommen werden.

## 11.1 Funktionsweise und Grundstruktur

Der Aufbau besteht aus einem lokalen Webserver namens Jetty<sup>10</sup> mit einer minimalen Weboberfläche ohne eigene Nutzdaten und einem entfernten Webserver mit Daten. Für Letzteren wurde aus der Liste der unterstützten Anwendungen die Software Bugzilla ausgewählt. Unter <http://landfill.bugzilla.org> steht ein öffentlicher Testserver zur Verfügung, zu dem vom lokalen Webserver eine Verbindung hergestellt wird. Dazu werden der lokalen Webanwendung die Zugangsdaten des entfernten Testservers per Konfiguration bekanntgegeben. Das lokale Webinterface stellt nur minimale Funktionen zur Suche und zum Erstellen neuer Tickets bereit. Der Verbindungsauflauf und die Übermittlung der Daten folgt der gleichen Methodik wie die Python-Bugzilla-Anbindung in Kapitel 9.3 *Python-Skript* auf Seite 48, weshalb hier auf neue Schaubilder verzichtet wird.

## 11.2 Verbindungsauflauf und Abfrage

Der Benutzer meldet sich lokal bei Jetty an <http://localhost:8080/> an und öffnet das Suchformular. Nach der Eingabe des Suchbegriffs und Drücken auf „Search“ wird die Verbindung zum Testserver von Bugzilla aufgebaut. Der Benutzer muss die dortigen Zugangsdaten eingeben. Lokal werden daraufhin die Tickets mit Nummern und Namen angezeigt, die das Suchwort im Betreff enthalten. Der Suchbegriff wird mittels des REST-Protokolls vom lokalen Server an den entfernten Server übertragen, woraufhin dieser die Anfrage auswertet und ebenfalls per REST die Daten an das lokale System zurücksendet. Das lokale System wertet das Ergebnis aus und stellt die gefundenen Ressourcen dar, ohne sie selbst lokal zu speichern.

Der Benutzer meldet sich am lokalen Webserver an und hat zunächst keinen Einblick in die Daten der angeschlossenen Systeme. Benutzt er eine der Funktionen, die Daten von einem anderen Service benötigen, wird nach erfolgreicher

---

<sup>10</sup><http://www.eclipse.org/jetty/>

Anmeldung ein Verbindungsaufbau initiiert und der Zugang zu den angeforderten Daten auf dem abgelegenen System gewährt.

Das Erstellen eines neuen Tickets läuft analog ab: per Knopfdruck auf „Create“ in der lokalen Anwendung schickt die lokale Webanwendung die Daten aus dem Formular per REST an den Bugzilla-Testserver. Dieser legt ein Ticket an, das daraufhin per dortiger Webanmeldung eingesehen werden kann. Wieder per Suche im lokalen System kann das neu angelegte Ticket auch lokal sichtbar gemacht werden.

### 11.3 Verbindungskomplexität zwischen REST-Anwendungen

Allein das Verwenden von REST mit der in *11.2 Verbindungsauflaufbau und Abfrage* beschriebenen Methode senkt noch nicht die Komplexität der Programmverbindungen untereinander. Wenn ein Benutzer beispielsweise in einem Testmanagementtool einen Defekt für einen Bugtracker anlegen will, ohne die Plattform zu verlassen, muss der Implementierer des Testmanagementtool ihm alle Felder und Zuordnungen samt Validierungen des Bugtrackers in seiner bestehenden Software anbieten. Das überfrachtet nicht nur die eigene Software, es ist auch ein großer Aufwand, hochgerechnet auf alle sinnvollen Verbindungen je zweier REST-Tools. Diese Methodik (publisher subscriber pattern – Beobachter-Entwurfsmuster[4]) sollte deshalb nur verfolgt werden, wenn es sich um überschaubar viele Schnittstellen handelt. Ist die Anzahl der Schnittstellen hoch, lohnt es sich womöglich eher, das mediator pattern[8] (Vermittler-Entwurfsmuster) einzusetzen und die Intelligenz in den Vermittler zu bringen, um mit möglichst wenigen Änderungen der anzuschließenden Systeme dennoch eine Vielzahl an Schnittstellen bereitzustellen.

Tabelle 12 zeigt für die Anzahl der Verbindungen, wie schnell der Aufwand steigt:

Knoten	Kanten	Schnittstellen
1	0	0
2	1	2
3	3	6
4	6	12
5	10	20
⋮	⋮	⋮
n	$\sum_{i=1}^{n-1} i = \frac{(n^2 - n)}{2}$	$2 * \sum_{i=1}^{n-1} i = (n^2 - n)$

Tab. 12: Knoten-abhängiges Wachstum der Schnittstellenanzahl

Für fünf Knoten liefert das bereits zehn Kanten mit bis zu 20 Schnittstellen,

wenn immer auf allen betroffenen Knoten implementiert werden muß. Die Werte sind dichte obere Schranken, im Einzelfall können weniger Implementierungen erforderlich sein. Die Anzahl notwendiger Stern-Verbindungen zu einem zentralen Hub (Vermittler) beträgt dagegen nur fünf Kanten von jedem Knoten zum Zentrum.

Ein Mittel gegen die Erweiterung der einen Anwendung um Funktionen der anderen ist anderer Integrationsstil von OSLC - das Konzept der Dialoge (Delegated user interface dialogs). Statt selbst einen Dialog für das Testmanagementtool zu entfernen, bittet der Implementierer das Bugtrackingtool, ihm zu diesem Zweck einen Dialog seines eigenen Systems (Bugtrackingtool) anzuzeigen. Das Testmanagementtool liefert bei der Anfragen die initialen Daten für das Erzeugen des Defekts und erhält zugleich den URL durchgereicht, den das Bugtrackingtool anlegen wird. Das ist, was in 11.4.2 *Vorschau per ninaCRM* gezeigt wird.

OSLC bietet dazu zwei Arten von Dialogen:

- Ressourcen erzeugen (Benutzer will neue Ressource in einem OSLC Service Provider anlegen): Anwendung fragt beim Provider ein User Interface zum Anlegen einer Ressource an; Provider benachrichtigt Anwendung, wenn die Ressource angelegt oder abgebrochen wurde
- Ressourcen auffinden (Benutzer will bestehende Ressource eines OSLC Service Providers auswählen): Anwendung fragt beim Provider ein User Interface zum Auffinden einer Ressource an; Provider benachrichtigt Anwendung, wenn die Ressource ausgewählt oder die Auswahl abgebrochen wurde

## 11.4 Vorschaufunktion für Bugzilla-Defekte

Außer der direkten Suche und Anzeige zu Details eines eingestellten Bugtickets erlaubt das Framework auch eine Vorschau in Form eines Tooltips als Mouseover-Effekt. Dafür muss keine explizite Suchanfrage abgesetzt werden. In einem REST-Client wird zudem der übermittelte Quellcode ersichtlich, der je nach Anforderung durch den Client anders aussieht und so gezielt ausgewertet werden kann.

### 11.4.1 Vorschau per REST-Client

Der folgende Quellcode-Auszug von Listing 21 zeigt die RDF-formatierte Antwort des Web Services mit Links zu einer kleinen und einer großen Vorschau für die Anfrage per REST-Client an <http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/1> mit dem Headertyp `application/x-oslc-compact+xml`. Die konsumierende Anwendung kann damit je nach Anforderung den einen oder anderen Ast auswerten und eine unterschiedlich detaillierte Vorschau bieten. Diese kann z.B. für Endgeräte mit kleinen Anzeigebereichen gezielt ausgewählt werden.

Nicht benötigte oder interpretierbare Äste kann die Anwendung dank des fehler-toleranten Dateiformats ignorieren, ohne Funktionalität einzuschränken oder zu verhindern.

Listing 21: Antwort eines Web Services im RDF-Format

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:oslc="http://open-services.net/ns/core#"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <oslc:Compact rdf:about="http://localhost:8080/
        OSLC4JBugzilla/services/1/changeRequests/1">
        <oslc:largePreview>
            <oslc:Preview>
                <oslc:hintHeight>20em</oslc:hintHeight>
                <oslc:hintWidth>45em</oslc:hintWidth>
                <oslc:document rdf:resource="http://localhost:8080/
                    OSLC4JBugzilla/services/1/changeRequests/1"/>
            </oslc:Preview>
        </oslc:largePreview>
        <oslc:smallPreview>
            <oslc:Preview>
                <oslc:hintHeight>11em</oslc:hintHeight>
                <oslc:hintWidth>45em</oslc:hintWidth>
                <oslc:document rdf:resource="http://localhost:8080/
                    OSLC4JBugzilla/services/1/changeRequests/1/
                    smallPreview"/>
            </oslc:Preview>
        </oslc:smallPreview>
        <dcterms:title>bugzilla bugzy 1</dcterms:title>
        <oslc:icon rdf:resource="https://landfill.bugzilla.org/
            bugzilla-4.0-branch//images/favicon.ico"/>
    </oslc:Compact>
</rdf:RDF>
```

Die kleine Vorschau in Abbildung 41 listet daraufhin nur wenige Felder auf.

Die große Vorschau in Abbildung 42 zeigt dagegen den kompletten Ticketinhalt, hier nur im Auszug.

#### 11.4.2 Vorschau per ninaCRM

Auf dem lokalen Webserver jetty ist ein minimales Webfrontend als Anschauung für eine Fremdanwendung mit REST-Schnittstelle, aber ohne eigene Datenhaltung



Abb. 41: kleine Vorschau - smallPreview

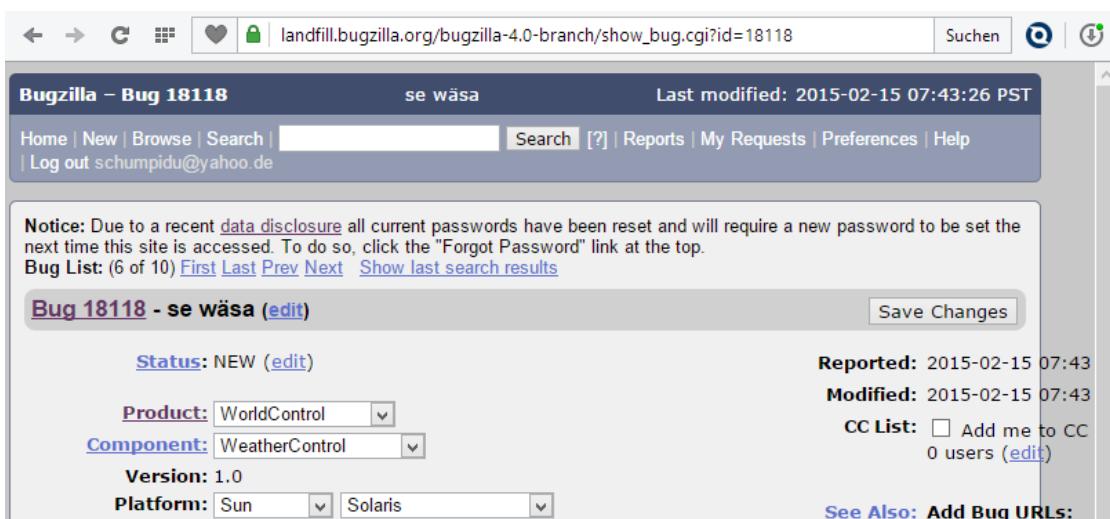


Abb. 42: große Vorschau - largePreview

installiert. Das lokale ninaCRM arbeitet als Proxy und kontaktiert im Bedarfsfall die erforderlichen Systeme mit den Echtdaten, um dem Benutzer die gewünschten Informationen anzuzeigen. In diesem Prototypszenario beschränkt sich ninaCRM auf die Anbindung an Bugzilla, es kann jedoch als Einstiegspunkt für weitere Systeme angesehen werden. Das Aufrufen der Seite <http://localhost:8181/ninacrm/> zeigt zunächst Statusinformationen für den Vorfall #676 und mit ihm in Beziehung stehende Defekte, siehe Abbildung 43.

Abb. 43: Standardansicht ohne Vorschau

Beim Fahren mit der Maus über den ersten Defekt wertet das JavaServer Pages-Formular die Aktion aus und versucht, die dort konfigurierte Vorschau mit den Daten für Bug #2 anzuzeigen, in diesem Fall `changerequest_preview_small.jsp`. Da die Defektinformationen für Bug #2 nicht auf dem lokalen Server gespeichert sind sondern auf <http://landfill.mozilla.org>, stellt BugzillaChangeRequest-Service.java die Verbindung mit dem entfernten Host her, um die Daten remote abzurufen. Der entfernt stehende Bugzilla-Server beantwortet die Kontaktaufnahme mit dem Erfordernis, sich am System anzumelden, wie in Abbildung 44 dargestellt.

Nach erfolgter Anmeldung werden beim Verweilen mit der Maus auf dem Defektlink die Informationen für die kleine Vorschau abgerufen, wie in Abbildung 45 dargestellt.

**Incident #676**

Status	Bitte melden Sie sich an Sie müssen sich mit »localhost:8181« anmelden Website-Meldung: Bugzilla	
Customer	Totally Fictional	Benutzername: <input type="text"/>
Created	Feb. 15, 2012	Passwort: <input type="password"/>
Updated	Feb 21, 2012	
Status	OPEN	<input type="button" value="Anmelden"/> <input type="button" value="Abbrechen"/>

Description: Lorem ipsum et cum fabulas indoctum consequuntur, te eum habeo eleifend. Usu cetero scribentur no, ius ad nominati accusamus accommodare. Dolorem appellantur te mel, nihil latine expetendis usu at, mel ei prima graeco. Harum scribentur est in. Mel cu natum interessel, suas menandri salutatus at est, debet ignota qui an. Epicurei scribentur ei pri. Cu utroque vituperata cum, agam invidunt ei nec, eum eu sonet possit.

[Add Link...](#) [Select Defect to Link to...](#) [Create Defect to Link to...](#)

**Related Defects**

- [Bug #2](#)
- [Bug #1](#)
- [Bug #8](#)

Abb. 44: Anmeldemaske beim Anfordern von Ticketdetails

**Incident #676**

Status		
Customer	Totally Fictional Corporation, Inc.	
Created	Feb. 15, 2012	
Updated	Feb 21, 2012	
Status	OPEN	

Description: Lorem ipsum et cum fabulas indoctum consequuntur, te eum habeo eleifend. Usu cetero scribentur no, ius ad nominati accusamus accommodare. Dolorem appellantur te mel, nihil latine expetendis usu at, mel ei prima graeco. Harum scribentur est in. Mel cu natum interessel, suas menandri salutatus at est, debet ignota qui an. Epicurei scribentur ei pri. Cu utroque vituperata cum, agam invidunt ei nec, eum eu sonet possit.

[Add Link...](#) [S...](#)

**Related Defects**

- [Bug #2](#)
- [Bug #1](#)
- [Bug #8](#)

**Status:** REOPENED      **Product:** FoodReplicator  
**Assignee:** tara@bluemartini.com      **Component:** SaltSprinkler  
**Priority:** P1      **Version:** 1.0  
**Reported:** 16.06.00 01:07      **Modified:** 19.09.14 09:58

Abb. 45: Erweiterte Ansicht mit Vorschau nach Authentifizierung

### 11.4.3 Diensterkennung

Ein Consumer startet mit dem Feststellen von angebotenen Diensten (Service Discovery). Er interagiert mit einem OSLC Service Provider, indem er zuerst per HTTP-GET dessen Provider Service Catalog abfragt (Retrieve). Dieser Katalog verweist auf Service Provider, welche die Verknüpfungen zu weiteren Katalogen oder Service Providern liefern, um OSLC Change Requests zu erzeugen oder abzufragen. Ein JavaServer Pages-Formular zeigt daraufhin den Katalog und die Providerinformationen an. Abhängig von den zur Verfügung gestellten Diensten ist es dem Consumer erlaubt, über eigene oder angeforderte Formulare Daten einzusehen oder neue zu erzeugen. Abbildung 46 veranschaulicht den Ablauf.

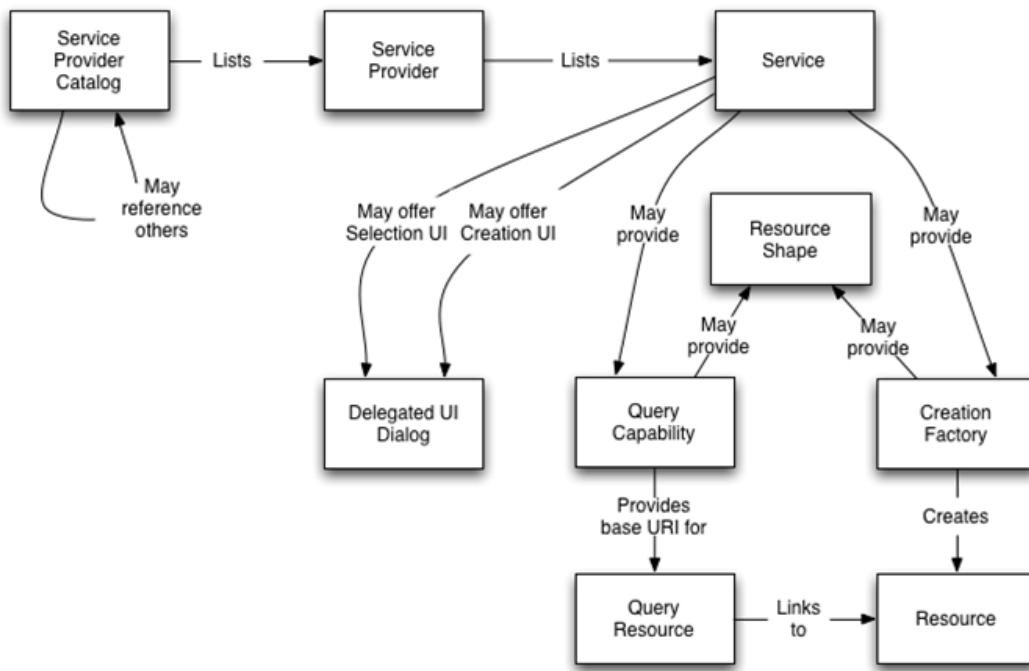


Abb. 46: Diensterkennung in OSLC [3]

## 12 Perspektiven für ein Anschlussprojekt

Die Vorarbeiten aus diesem Projekt bieten den Einstieg in das semantische Web und die Welt der Ontologien als Daten- und Organisationsstruktur für einen integrierten Entwicklungsprozess. Der Ansatz ist vielversprechend und bietet genug

Potential zum Vertiefen der hier gewonnenen Erkenntnisse mit dem Ziel einer heterogenen IT-Landschaft unter dem Dach eines universellen, erweiterbaren und lernfähigen Wissenssystems. Mögliche, auf diesem System aufsetzende, Arbeiten sind:

- Evaluation alternativer Werkzeuge zum Modellieren der Ontologie mit Fokus auf:
  - Stabilität des Hauptwerkzeugs und der Plugins
  - schnelleren/besseren *Reasonern*, ggf. Selbstanpassung<sup>11</sup>
  - Möglichkeiten der Echtzeitauswertung
  - graphischen Darstellung der mit *SPARQL* erstellten Abfragen
  - feinerem Ein-/Ausblenden von Knoten und Kanten im Ontographen, ggf. Selbstanpassung
- striktere Benutzung und Auslegung der *Domains*, *Ranges*, *Cardinalities*, *Functionals*, um auf Abfragen präzisere Auskünfte von der Ontologie zu erhalten
- Ausnutzen der weiteren Sprachmöglichkeiten von *SPARQL* zum Verketten von Abfragen und Ändern von Daten
- Analyse und Integrationsplan der per REST anschließbaren Plattformen
- Analyse und Integrationsplan der nicht per REST anschließbaren Plattformen: können diese nachgeRESTet werden? Ist eine Einbindung des Python-Skripts oder einer ähnlichen Technologie mit vertretbarem Aufwand möglich? Können Konvertierprogramme für diese Datenformate geschrieben werden (für MS Office-basierte Dateiformate, CAD/CAM-Systeme und weitere)? Wie wird mit Anwendungen verfahren, die keine Konvertierung der Daten erlauben?
- umfassende Definition des gemeinsamen Namensraums der anzuschließenden Plattformen
- Ausbau des von Protégé erzeugten Java-Codes
- Generieren von Schnittstellen (Pseudo-Methodenaufrufen) aus den *SPARQL*-Abfragen für die Definition des Austauschs von Daten unterschiedlicher Systeme

---

<sup>11</sup>Bewertung weiterer Reasoner, z.B. RacerPro (kommerziell; für universitären Bereich gratis) als Reasoner für Protégé: Siehe <https://www.ifis.uni-luebeck.de/index.php?id=385>, User's Guide <http://www.ifis.uni-luebeck.de/~moeller/racer/Racer-2.0/users-guide-2-0.pdf>, Kapitel 2.4.2

- Vernetzen der Nutzdatensysteme mit dem Ontologie-führenden System
- Ausweiten des Ontologie-Vokabulars auf die anzuschließenden Systeme
- Abstrahieren der begonnenen OSLC-Integration zur allgemeinen Anbindung von RESTful-fähigen Systemen
- Ausbau des Lyo-/OSLC-Frameworks zur sukzessiven Anbindung weiterer Systeme
- Verwenden des JENA-Frameworks<sup>12</sup> zum Anbinden von Datenbanksystemen an die Ontologie
- gezieltes Anbieten von Diensten für einzelne Anwendungen zur Einsichtnahme von Daten aus anderen Systemen
- Aufsetzen einer zentralen WebProtégé-Instanz als unternehmensweites Wissenssystem
- Etablieren eines Rechtesystems als Zugangskontrolle zu Subdomänen der Ontologie
- Definition einer Verfahrensweise zur Integration bestehender Systeme in die Ontologie-Landschaft mit Hinblick auf
  - einfache und universelle Erweiterbarkeit
  - Erfüllen von Qualitätsstandards der ISO-Norm 250XX wie ISO/IEC 25010:2011[1]

Die Vielfalt dieser Aufgaben zeigt, welch mächtiges Instrument ein Wissenssystem darstellt und in welchen Bereichen es überall genutzt werden kann. Dabei gehen die Möglichkeiten weit über Produktentwicklungsprozesse hinaus. Sowohl in kleineren als auch komplexeren Systemen ist eine Kopplung der Systeme unterschiedlicher Anbieter erstrebenswert, um den Fluss von Informationen nicht nur auf dem Papier sondern auch EDV-gestützt zu kanalieren und beschleunigen. Dafür gibt es keine Universallösung - auch eine Ontologie ist keine. Für manche Anwendungen sind andere Ansätze wie Datenbanken oder Neuronale Netze die geeigneter Wahl. Ein großer Vorteil einer Ontologie ist jedoch, dass mit überschaubarem Aufwand ein Grundgerüst für ein heterogenes, aber integriertes und auch integratives Wissenssystem gebaut werden kann. Die weiteren Arbeiten können nahezu beliebig parallelisiert oder sequentialisiert werden, da viele der genannten Erweiterungsmöglichkeiten nicht zwingend aufeinander aufbauen. Sie ergänzen

---

<sup>12</sup><http://jena.apache.org/>

sich vielmehr in der Breite der Anwendungen für die Unternehmens- oder Organisationsbereiche, die sukzessive daran angeschlossen werden. Technologien wie REST, SOAP und WDSL, die nach und nach in immer mehr Anwendungen Einzug halten, helfen dabei und geben Systemverwaltern leistungsfähige Hilfsmittel an die Hand, um den Arbeitsaufwand pro neu aufzunehmendes System zu senken und damit auch den Schulungsaufwand zu verringern, weil sich die Entwickler auf wenige universelle Verfahren konzentrieren können.

Die Idee solcher Sprachen und Technologien an sich ist nicht neu: Interprozesskommunikationssprachen wie Rexx<sup>13</sup> und Datenaustauschsprachen wie Rebol<sup>14</sup> erlauben seit Jahrzehnten den Austausch von Daten über Anwendungsgrenzen hinweg. Häufig waren diese jedoch pro Betriebssystem angepasst und wenig auf die Kommunikation zwischen Systemen ausgerichtet oder erlangten nicht die breite Masse der Anwendungen, weshalb sie nur in Nischen intensiv genutzt wurden. Neuere Technologien wie REST gehen einen anderen Weg. Durch die zunehmende Anzahl an Anwendungen und die einfache und offene Schnittstelle haben sie die Chance, zu Standards heranzureifen. Schon bei Rexx war ein wichtiges Konzept, dass jeder Wert durch einen String repräsentiert wurde - ein Konzept, das für Ontologien im Grundsatz wieder aufgegriffen wurde, auch wenn weitere Datentypen erlaubt sind.

## 12.1 Schlusswort

Trotz der teils prinzipbedingten, teils programmatischen Einschränkungen in der Modellierungsphase wird die Fortführung dieses Projektes empfohlen. Das Ziel einer Wissensrepräsentation, die prozessübergreifend nutzbar ist und somit die Kommunikation und Verknüpfung einzelner Prozessschritte innerhalb des Innovationsprozesses ermöglicht, wurde mit Hilfe der Erstellung einer Ontologie erreicht. Trotz fehlender Einhaltung einiger Restriktionsangaben durch den *Reasoner* schafft die Ontologie einen einheitlichen Wissensrahmen und trägt damit wesentlich zur Komplexitätsreduktion der Kommunikation innerhalb eines Unternehmens bei. Die Modellierung eines Beispiels beweist die Möglichkeit der Datenpropagation durch die Verbindung von *Individuals* über *Object Properties*. Diese können zusätzlich zur Modellierung von Schnittstellen zwischen unterschiedlichen Unternehmensabteilungen genutzt werden. Das Beispiel kann beliebig erweitert und neue Abteilungen und Prozesse nach individuellen Bedürfnissen eingefügt und in die bisher bestehende Unternehmenslandschaft integriert werden. Auch der Prototyp, der mit Hilfe des REST- und OSLC-Frameworks umgesetzt wurde, erfüllt die erwartete Vereinheitlichung der Kommunikation und eine erleichterte Schnittstellenverwendung.

---

<sup>13</sup><http://www-01.ibm.com/software/awdtools/rexx/>, <http://en.wikipedia.org/wiki/Rexx>

<sup>14</sup><http://www.rebol.com>

Alle in diesem Projekt verwendeten Programme und Schnittstellenmodellierungen sind quelloffen, worin eine gute Möglichkeit zur individuellen Anpassung besteht. Damit legte dieses Projekt einen Grundstein, der durch einen ausführlichen Überblick über die Datenmodellierungs- und Integrationsprinzipien ein Gerüst bildet, auf dem nachfolgende Projekte erfolgreich aufbauen können.

## 13 Glossar

<i>Annotation</i>	Unter 'Annotation' kann eine Beschreibung zu allen eingetragenen Klassen, Properties, usw. eingefügt werden.
<i>Cardinality</i>	Die Kardinalitäten müssen bei allen Object Properties eingetragen werden. Sie geben an, zu wie vielen Individuals einer anderen Klasse ein Individual dieser Klasse in der jeweiligen Beziehung stehen darf.
<i>Class</i>	'Class' beschreibt die Gegenstände der Umgebung, die die Ontologie darstellt. Durch das Erstellen von Unterklassen entsteht eine Hierarchie. Die Unterklassen erben die Eigenschaften und Beziehungen der Oberklasse.
<i>Class expression editor</i>	Subclasses können durch Anklicken der entsprechenden Object Property und der Class gebildet werden. Es ist aber auch möglich, komplexe Verknüpfungen bzw. Restriktionen per Hand mit Hilfe des 'Class expression editor' einzutragen.
<i>Data Property</i>	In den 'Data Properties' werden Attribute der Klassen mit dem jeweiligen Datentypen eingetragen.
<i>Data property assertion</i>	Mittels der 'Data property assertions' werden die Eigenschaften der Individuals mit den jeweiligen Datentypen definiert.
<i>Domain, Range</i>	Hier lassen sich der Definitions- und der Wertebereich der Properties eintragen.
<i>Equivalent To</i>	'Equivalent To' beschreibt die notwendigen Bedingungen einer Beziehung oder Klassen mit den selben Eigenschaften.
<i>Individual</i>	Individuals stellen die existierenden Dinge dar. Sie sind Instanzen der Klassen.
<i>Inverse Of</i>	Mit 'Inverse of' lassen sich Object Properties verbinden, die jeweils die Gegenrichtung einer Beziehung beschreiben.
<i>Object Property</i>	Um Beziehungen zwischen verschiedenen Klassen darzustellen, gibt es 'Object Properties'. Hier können auch die jeweiligen Kadinalitäten eingefügt werden.
<i>Object property assertion</i>	Unter 'Object property assertions' können Beziehungen zwischen zwei Individuals hergestellt werden.
<i>OntoGraf</i>	Im OntoGraf lassen sich die Klassen und Individuals mit den jeweiligen Beziehungen grafisch darstellen.
<i>Reasoner</i>	Der Reasoner überprüft die Ontologie auf Konsistenz und schlussfolgert neues Wissen.

*SPARQL*

Die Abfragesprache in Protégé nennt sich SPARQL (SPARQL Protocol And RDF Query Language). Damit lassen sich Individuals erstellen, aber auch Anfragen an die Ontologie stellen.

*Subclass Of*

Unter 'Subclass of' werden die hinreichenden Bedingungen für Beziehungen oder Oberklassen für Klassen eingetragen.

*URI*

Jede Ontologie, aber auch die Klassen, haben einen eigenen Uniform Resource Identifier. Bei SPARQL-Abfragen muss der URI der Ontologie als Präfix angegeben werden.

## 14 Namenskonventionen

- Alle Bezeichner der Ontologie werden mit englischen Namen benannt. Umlaute kommen entsprechend nicht vor.
- Substantive wie Seriennummer, Version, Tool etc. werden wie deutsche Substantive mit erstem großen Buchstaben geschrieben.
- Adjektive und Verben werden komplett klein geschrieben.
- Wörter werden durch Unterstrich getrennt.
- Problematische Sonderzeichen wie Et-Zeichen (ugs. „Kaufmannsund“) werden umschrieben („and“) oder durch Bindestrich ersetzt.

### 14.1 Abkürzungsverzeichnis

Die folgenden Abkürzungen werden als Suffixe an die Bezeichner der Ontologie angehängt, um deren Herkunft zu verdeutlichen.

#### 14.1.1 Prozesse

CD	Customer Discovery
PL	Planning
PD	Product & Factory Development
TE	Test
CF	Customer Feedback
CV	Customer Validation
CC	Customer Creation
CB	Company Building

#### 14.1.2 Dokumente

RP	Release Plan
RQ	Requirements
BM	Business Model
RS	Resources
CI	Customer Information
WP	Work Packages
PF	Product / Factory
RA	Reports / Analysis

## 15 Klassendokumentation mit OWLDoc

Das Plugin OWLDoc für Protégé erlaubt den Export der Klassen und Individuen einer Ontologie in einer Javadoc-artigen Darstellung. Damit werden HTML-Dokumente generiert, die als externes Dokumentationssystem genutzt und z.B. im Intranet bereitgestellt werden können, um auch Entwicklern ohne Zugang zu Protégé oder dem eingesetzten Modellierungswerkzeug einen Einblick in die Definitionen und Verknüpfungen innerhalb der Ontologie zu erlauben. Beispielhaft sei hier an der *Data Property Type* gezeigt, in welchen *Domains* und *Ranges* sie Verwendung findet und welche Ausprägungen es davon gibt.

The screenshot shows a two-column interface. The left column, titled 'Contents', lists categories such as 'aws', 'Classes (132)', 'Object Properties (35)', etc., and a detailed list of individuals under 'Renting\_or\_Lending\_or\_Leasing'. The right column, titled 'Data Property: Signal', provides detailed information about the property, including its annotations, superproperties, and usage examples. A link to the characteristic class is also present.

<b>Contents</b>	<b>Data Property: Signal</b>
<ul style="list-style-type: none"><li>aws</li><li>Classes (132)</li><li>Object Properties (35)</li><li>Data Properties (123)</li><li>Annotation Properties (1)</li><li>Individuals (99)</li><li>Datatypes (4)</li></ul>	<p><b>Annotations (1)</b></p> <ul style="list-style-type: none"><li>comment "a sign with a certain meaning/display; signal shape" (string)</li></ul>
	<p><b>Superproperties (1)</b></p> <ul style="list-style-type: none"><li>Characteristic</li></ul>
	<p><b>Usage (3)</b></p> <ul style="list-style-type: none"><li>Input_Signal <math>\sqsubseteq</math> Signal</li><li>Output_Signal <math>\sqsubseteq</math> Signal</li><li>Signal <math>\sqsubseteq</math> Characteristic</li></ul>
	<p><a href="http://www.semanticweb.org/aws#Characteristic">http://www.semanticweb.org/aws#Characteristic</a></p>

Abb. 47: Ausschnitt aus dem Aufbau von OWLDoc

Da OWLDoc statische Seiten erzeugt, veraltet die Dokumentation nach jeder Änderung am Modell und muss neu aus Protégé exportiert werden. Der Export des vorliegenden Modells dauert nur etwa eine Minute. Da der Quellcode von Protégé frei erhältlich ist, sollte sich mit wenigen Anpassungen ein automatischer, zeitgesteuerter Export per Cronjob erstellen lassen. Auf diese Weise kann ohne manuellen Eingriff nach jedem Speichern des Modells ein automatischer Build-Prozess ausgelöst oder in regelmäßigen Abständen eine aktualisierte Dokumentation erzeugt werden. Ein Erzeugen der Dokumentation aus der XML-Datei statt aus Protégé sollte ebenfalls mit wenigen Änderungen möglich sein.

## Literatur

- [1] ISO/IEC 25010:2011, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=35733](http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733).
- [2] Alexander Alexandrov. Zusammenkunft mit Alexander Alexandrov vom Fachbereich DIMA, um die Syntax und Semantik von SPARQL zu erlernen und zu verstehen, 13. Februar 2015 im Raum EN 727.
- [3] Dr. Jose María Alvarez-Rodríguez. Embedded Systems OSLC. <http://slides.com/josemariaalvarez/2-embedded-systems-oslc-rm/embed>.
- [4] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
- [5] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.
- [6] Dr. Ing. Stefan Fricke. Persönliches Gespräch mit Dr. Ing. Stefan Fricke im Raum TEL1116, 07. Januar 2015, 11 Uhr.
- [7] Dr. Ing. Stefan Fricke. Persönliches Gespräch mit Dr. Ing. Stefan Fricke im Raum TEL1116, 12. Januar 2015, 10 Uhr.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [9] Otto Kerner, Joseph Maurer, Jutta Steffens, Stefan Thode, and Rudolf (Erarb.) Voller. *Vieweg-Mathematik-Lexikon: Begriffe/Definitionen/Sätze/Beispiele für das Grundstudium (German Edition)*. Vieweg+Teubner Verlag, 1 edition, 1988.
- [10] Graham Klyne, Jeremy J. Carroll, and Brian McBride. RDF 1.1 Concepts and Abstract Syntax. <http://www.w3.org/TR/rdf11-concepts/>, 25. Februar 2014.
- [11] Christoph Meinel and Martin Mundhenk. *Mathematische Grundlagen der Informatik: Mathematisches Denken und Beweisen. Eine Einführung*, 5. Auflage. Vieweg+Teubner, Wiesbaden, 5 edition, 2011 2011.

- [12] Alexander Osterwalder and Yves Pigneur. *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. Wiley Desktop Editions Series. John Wiley and Sons, 2010.
- [13] Renate Stücka. IBM Software Group, OSLC: Offener Standard für die Tool-Integration. <http://www.heise.de/developer/artikel/OSLC-Offener-Standard-fuer-die-Tool-Integration-1261465.html>, 21. Juni, 2011.
- [14] Bugzilla Team. Bugzilla REST-API. [https://wiki.mozilla.org/Bugzilla:REST\\_API](https://wiki.mozilla.org/Bugzilla:REST_API).

## 16 Anhang

## 17 Implementierungshandbuch

In diesem Abschnitt werden technische Details zum Einrichten des OSLC-/Lyo-Frameworks vorgestellt. Da die Konfiguration und Inbetriebnahme nicht geradeaus verliefen, sind hier einige Erläuterungen nötig.

### 17.1 Lyo-/OSLC-Framework

Das Lyo-/OSLC-Framework ist ein Software Development Kit für die Entwicklungsumgebung Eclipse zum Erstellen von Anwendungen des Lebenszyklusmanagements mittels einheitlicher Schnittstellen wie REST.

#### 17.1.1 Voraussetzungen für die Inbetriebnahme

Zu Beginn des Projekts gingen mehrere Wochen dadurch verloren herauszufinden, welche Versionen von Eclipse, maven, EGit und den Plugins zueinander kompatibel sind. Häufig ließen sich Pakete nicht installieren oder kompilieren, weil Bestandteile aus den Repositorys nicht heruntergeladen wurden oder mit anderen Paketen im Konflikt standen, die Lösungshinweise zu diesen Konflikten aber nicht halfen, der Webserver jetty wegen fehlender Konfigurationen kein Logging erlaubte und somit Hinweise auf mögliche Fehler verschwieg oder die Reihenfolge der Schritte in den Dokumentationen nicht stimmte und damit den Kompiliererfolg verhinderte. Diese Informationen lieferten die Tutorials nicht, weshalb die Suche nach geeigneten Einstellungen und Konfigurationen viel Zeit kostete. Eine konkrete Versionskombination war aus den Tutorials und von den Entwicklern nicht zu erhalten. Die Tutorials sind an vielen Stellen lückenhaft oder führen die nötigen Schritte in falscher Reihenfolge auf, so dass Module nicht kompiliert werden können, weil laut Dokumentation oder Entwicklerhinweisen abhängige Module erst später gebaut werden sollen, siehe unten am Beispiel der OAuth-Module. Teilweise beinhalten die Anleitungen Sprünge zu anderen Modulen, die nur durch mühsames Herumprobieren herausgefunden werden konnten. Letztlich konnte durch viel Zeitaufwand eine funktionierende Konfiguration gefunden werden. Die nötigen Hinweise für die Änderungen an der Dokumentation ist den Entwicklern zugegangen und soll in künftigen Versionen berücksichtigt werden.

Von den Entwicklern bei IBM war zu erfahren, dass Java 8 bislang nicht unterstützt wird, obwohl es anfänglich zu funktionieren schien. Ebenso wurde davon abgeraten, das Repository lokal in einen Pfad mit Leerzeichen abzulegen, nachdem auch hier Fehler das Kompilieren verhinderten. Nach dem Verschieben und Neueinrichten plus Installieren der maven-Pakete war das Setup funktionstüchtig.

### 17.1.2 Konfiguration für das Projekt

Die folgende Konfiguration wurde für das Framework verwendet:

- Windows 8.1/Windows XP
- Java 7, JDK 1.7.0\_76
- Eclipse Luna 1a (4.4.1) (Apache maven schon integriert: m2e 1.5.0.20140606-0033)
- EGit 3.4.2.201412180340-r
- keine weiteren Plugins

Wegen der oben genannten Startschwierigkeiten ist diese Konfiguration als Beispiel zu sehen. Das Framework kann mit anderen Konstellationen ebenfalls funktionieren. Es gab jedoch mehrere Kombinationen, in denen das nicht der Fall war. Für dieses Projekt wurde die genannte Konfiguration verwendet.

Der Quellcode für das Python-Skript und Lyo-Framework wurde in einem git-Repository abgelegt. Zugriff darauf kann auf Anfrage erteilt werden.

### 17.1.3 Einrichtung in Eclipse

An dieser Stelle soll keine komplett neue Dokumentation für das Lyo-Framework entstehen, das leisten im Wesentlichen die Webseiten des OSLC-Projekts. Hier werden nur wesentliche Punkte und Abweichungen von der Dokumentation genannt, die erfüllt werden mussten, um das Framework für dieses Projekt funktionstüchtig zu machen. Zu Beginn der Teilaufgabe OSLC bezog sich die offizielle Dokumentation noch auf Bugzilla 4.0. Kurz vor Fertigstellung dieses Kapitels wurde die Webseite <http://open-services.net/> und das Tutorial auf <http://open-services.net/resources/tutorials/integrating-products-with-oslc/running-the-examples/> bereits überarbeitet, so dass es mittlerweile Bugzilla 4.2 adressiert. Eventuell dadurch erforderliche Änderungen werden hier nicht berücksichtigt.

Lyo-Repositories klonen mit jeweils allen Unterästen:

- <http://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.core.git>
- <http://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.docs.git>
- <http://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.server.git>

Bugzilla-Einrichtung:

- Bugzilla-Account auf <http://landfill.mozilla.org> erstellen; in diesem Projekt wurde <https://landfill.bugzilla.org/bugzilla-4.0-branch/> verwendet
- `bugz.properties`-Dateien konfigurieren mit der auf <http://landfill.mozilla.org> registrierten Mailadresse und dem URI für die Bugzilla-Version
  - `bugzilla_uri=https://landfill.bugzilla.org/bugzilla-4.0-branch/`
  - `admin=<Mailadresse aus der Account-Registration>`

Die Authentifizierung im OSLC-Framework erfolgt über OAuth<sup>15</sup>, einem Protokoll zur sicheren Anmeldung über Webanwendungen. Dafür müssen folgende Komponenten mit Maven gebaut werden:

- `org.eclipse.lyo.server.oauth.consumerstore`
- `org.eclipse.lyo.server.oauth.core`
- `org.eclipse.lyo.server.oauth.webapp`

Entgegen der Dokumentation und Entwicklerauskünften mussten diese Pakete in folgender Reihenfolge übersetzt werden, anderenfalls war das Framework nicht funktionstüchtig:

- `org.eclipse.lyo.server.oauth.core`
- `org.eclipse.lyo.server.oauth.consumerstore`
- `org.eclipse.lyo.server.oauth.webapp`

Mit Maven musste zuerst die Grundkonfiguration pro Verzeichnis erstellt werden, ehe aus den abhängigen Paketen installiert werden konnte:

Paket `org.eclipse.lyo.server.oauth.core` öffnen

OSLC4j kompilieren: `pom.xml` -> Kontextmenü -> maven clean; `pom.xml` -> Kontextmenü -> maven install => BUILD SUCCESS.

LabMiscellaneous kompilieren: `pom.xml` -> Kontextmenü -> maven install => BUILD SUCCESS.

Zu diesem Zeitpunkt waren die weiteren Ausbaustufen der lokalen Webanwendung, die so genannten Labs, noch nicht kompilierfähig. Nach dem Verschieben des lokalen Repositorys in einen Leerzeichen-freien Pfad, nochmaligen Zurücksetzen der Grundkonfiguration und erneuten Installieren war das Framework einsatzbereit.

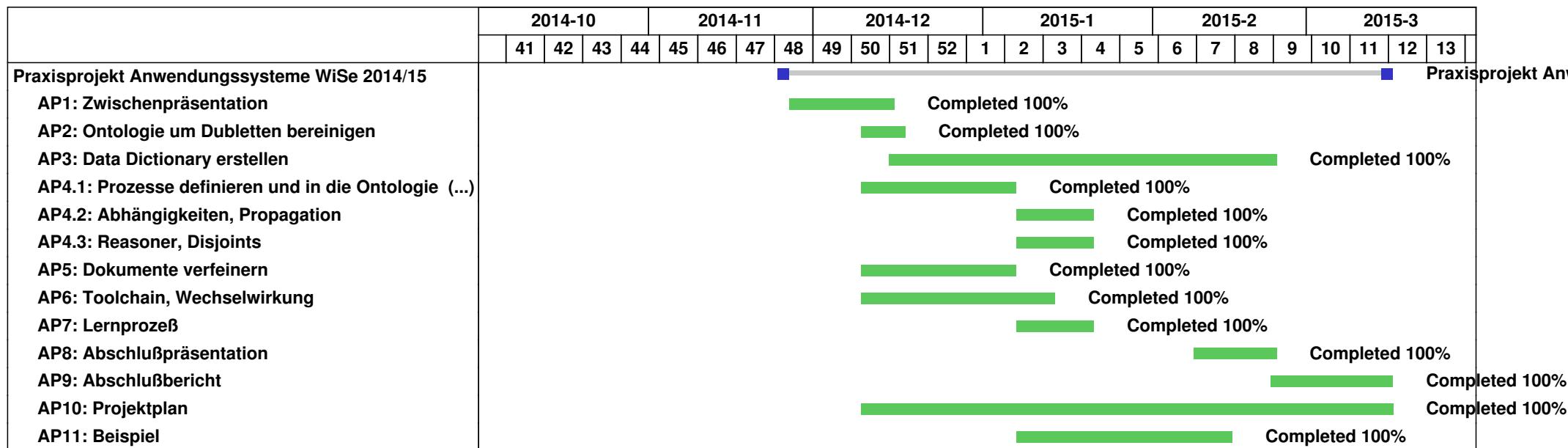
---

<sup>15</sup><http://oauth.net/>

## 18 Matrix

Prozesse / Daten	Release Plan	Requirements	Business Model	Resources	Customer Information	Work Packages	Product/Factory	Reports/Analysis
Customer Discovery			Value Proposition, Customer Segments, Customer Relationships		Verwendungszweck, Ausstattung, Geometrie, Ergonomie, Technologie			Market sharing, competitive abilities
Customer Validation		Identifikationsnr., Merkmal, Name der Anforderung, geforderte Ausprägung, zusätzliche Informationen, Verantwortlicher pro Anforderung, Bezeichnung, Zahlenwert(mit Toleranz), Einheit(phys.), Datum, Änderungsverfolgung	Key Activities, Key Resources, Value Proposition, Customer Segments, Customer Relationships		Ergebnisse von: Verwendungszweck, Ausstattung, Geometrie, Ergonomie, Technologie			Degree of satisfaction, Feedback, etc.
Planning		Identifikationsnr., Merkmal, Name der Anforderung, geforderte Ausprägung, zusätzliche Informationen, Verantwortlicher pro Anforderung, Bezeichnung, Zahlenwert(mit Toleranz), Einheit(phys.), Datum, Änderungsverfolgung	Cost structure, Key Activities, Key Resources, Customer Relationships, Customer Segments	<b>Sachressourcen</b> (Gebäude, Maschinen,...); <b>Finanzen</b> (Investoren, Kredite...)	Festlegung von: Verwendungszweck, Ausstattung, Geometrie, Ergonomie, Technologie	Identifikator, Projektinformation, Aufgaben, Ziele, Zeit, beteiligte Personen, Material, Kosten, Risiken, Dokumente, Sonstiges		All
Product & Factory Development		Identifikationsnr., Merkmal, Name der Anforderung, geforderte Ausprägung, zusätzliche Informationen, Verantwortlicher pro Anforderung, Bezeichnung, Zahlenwert(mit Toleranz), Einheit(phys.), Datum, Änderungsverfolgung	Key Resources, Key Activities, Cost Structure	<b>Sachressourcen</b> (Gebäude, Maschinen,...); <b>Technologie</b> ; <b>Knowhow</b> (Fachkräfte)		Identifikator, Projektinformation, Aufgaben, Ziele, Zeit, beteiligte Personen, Material, Kosten, Risiken, Dokumente, Sonstiges		Weak points, Market sharing, Benefit, Maintenance
Test	<b>Sachressourcen</b> (Anlagen); <b>Finanzen</b> (Testbudget); <b>Human Resources</b> (Arbeitstage)	Identifikationsnr., Merkmal, Name der Anforderung, geforderte Ausprägung, zusätzliche Informationen, Verantwortlicher pro Anforderung, Bezeichnung, Zahlenwert(mit Toleranz), Einheit(phys.), Datum, Änderungsverfolgung	Key Resources, Key Activities, Cost Structure, Value Proposition			Identifikator, Projektinformation, Aufgaben, Ziele, Zeit, beteiligte Personen, Material, Kosten, Risiken, Dokumente, Sonstiges	-Prototyp -fertige Produkte -virtueller Prototyp-Prozesse und Fertigungsanlagen	Prototyp reporter, Situation of equipments
Customer Feedback	Zusatzbedarfe für Testing: <b>Sachressourcen</b> (Anlagen); <b>Finanzen</b> (Marketingbudget); <b>Human Resources</b> (Arbeitstage)		Customer Segments, Customer Relationships, Value Proposition		Verwendungszweck, Ausstattung, Geometrie, Ergonomie, Technologie, Fragebogen	Identifikator, Projektinformation, Aufgaben, Ziele, Zeit, beteiligte Personen, Material, Kosten, Risiken, Dokumente, Sonstiges		All
Customer Creation	<b>Finanzen</b> (Marketingbudget); <b>Human Resources</b> (Arbeitstage)		Key Activities, Key Resources, Cost Structure, Customer Segments, Customer Relationships, Value Proposition		Kundensegmente			Market sharing, Competitive abilities, Degree of satisfaction, Market research
Company Building			Key Partners, Key Activities, Key Resources, Channels, Revenue Streams	<b>Organisation; geistiges Eigentum</b> (Patente); <b>Knowhow</b> (Fachkräfte); <b>Finanzen</b> (Investoren, Kredit)	Kundensegmente			Simulation reporter, Analyse reported different domains; Best practices

## Praxisprojekt Anwendungssysteme WiSe 2014/15



# Praxisprojekt Anwendungssysteme WS14/15

---

ABSCHLUSSPRÄSENTATION

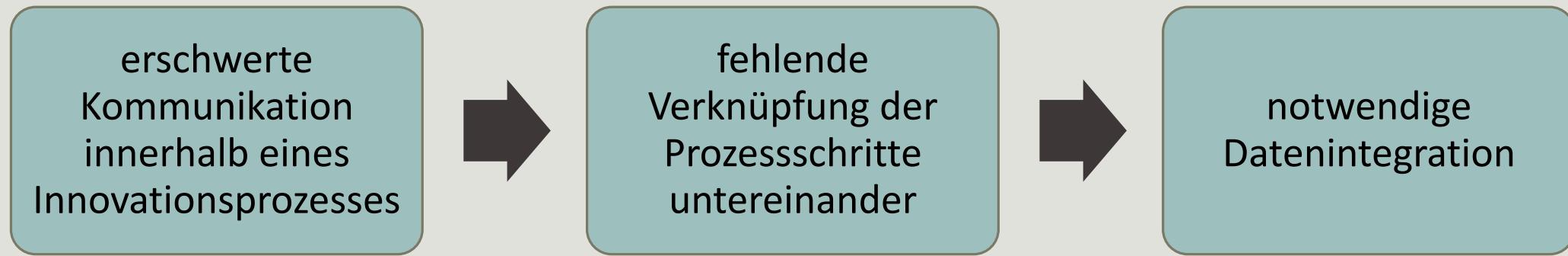
# Gliederung

---

1. Problemstellung
2. Lösungsansätze
3. Ontologie mit Protégé
4. Herausforderungen
5. Toolchain: REST, OSLC
6. Perspektiven

# Technische Umgebung für standardisierten, prozessübergreifenden Informationsaustausch fehlt

---



Vier potentielle Lösungsansätze wurden betrachtet

---

Datenbank

neuronales Netzwerk

Hybridlösung: DB + Individualsoftware

Ontologie

Eine Ontologie ist der beste Lösungsansatz, da sie intelligent und erweiterbar ist

---

Datenbank

neuronales Netzwerk

Hybridlösung: DB + Individualsoftware

Ontologie

# Ontologie

---

1

Modellierung der Ontologie

2

Ausarbeitung eines Beispiels

3

SPARQL-Abfragen

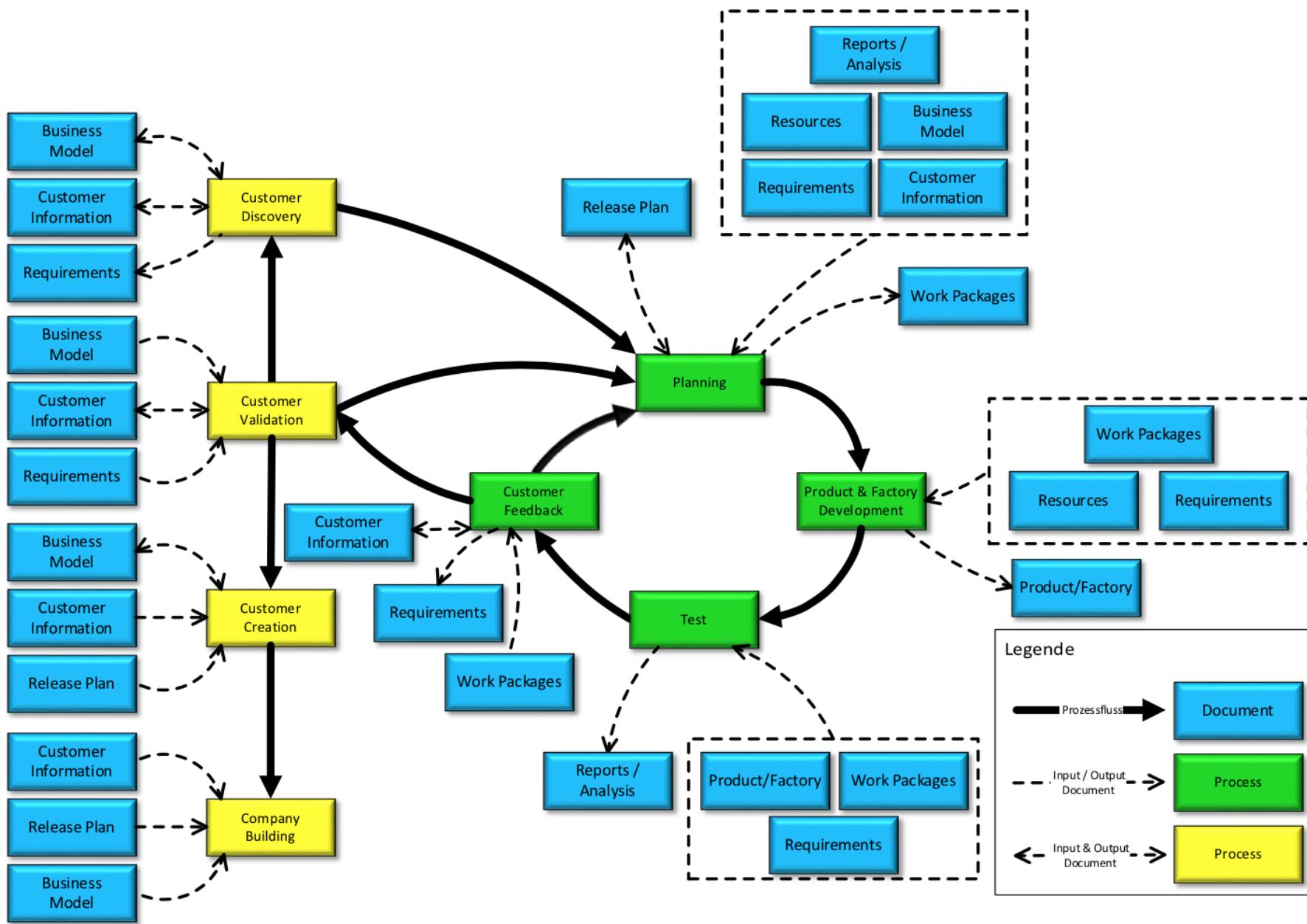


Abbildung 1: BÜTTNER, Florian (2014): Agiles Produkt- und Geschäftsmodell, unveröffentlicht

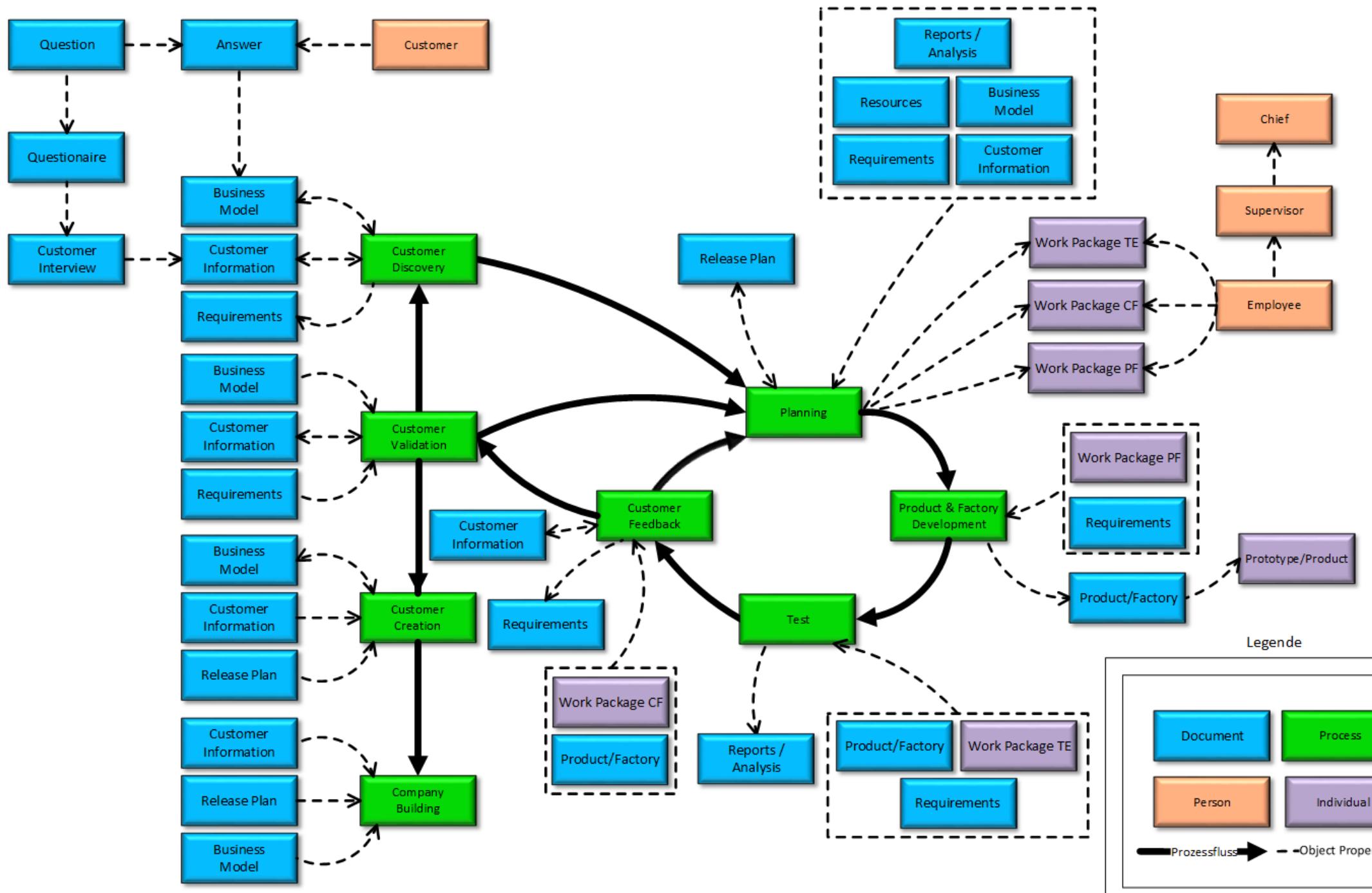
# Modellierung der Ontologie mit Hilfe des Editors Protégé

The screenshot shows the Protégé ontology editor interface. The top menu bar includes tabs for Active Ontology, Entities, Classes, Object Properties, Data Properties, Annotation Properties, Individuals, OWLViz, DL Query, and OntoGraf. The Entities tab is selected.

The left panel displays the Class hierarchy under the heading "Class hierarchy: Report\_and\_Analysis". It lists several classes, with "Report\_and\_Analysis" highlighted in blue. Other listed classes include Customer\_Segment\_BM, Description\_RQ, Hypothesis\_CI, Key\_Activity\_BM, Key\_Metric\_BM, Key\_Partner\_BM, Key\_Resource\_BM, Lean\_Canvas\_BM, Problem\_BM, Product\_and\_Factory, Question\_CI, Questionnaire\_CI, Release\_Plan, Release\_RP, Report\_RA, Requirement, Resource, Revenue\_Stream\_BM, Solution\_BM, Tracking\_of\_Modification\_RQ, and unfair\_Advantage\_BM.

The right panel displays the Annotations tab under the heading "Annotations: Report\_and\_Analysis". It shows an annotation for "rdfs:comment" with the value "document to gain a better understanding or knowledge of something". Below this, the Description tab is shown under the heading "Description: Report\_and\_Analysis". It lists properties such as "Equivalent To", "SubClass Of", and "Version".

At the bottom of the interface, there is a message: "To use the reasoner click Reasoner->Start reasoner" and a checked checkbox labeled "Show Inferences".



## Legende

# Die Arbeit mit Protégé stellte auf verschiedenen Gebieten eine Herausforderung dar

---

Modellierung	Reasoner	OntoGraf & SPARQL
<ul style="list-style-type: none"><li>• Positionierung der Restriktionen nicht intuitiv</li><li>• fehlende Dokumentation zu den Verknüpfungsoperatoren</li><li>• explizite Deklaration nicht gewünschter Eigenschaften notwendig</li></ul>	<ul style="list-style-type: none"><li>• keine Einhaltung aller gewünschten Restriktionen</li><li>• unzureichende Fehlermeldungen</li><li>• Pellet: bestmögliche Rückmeldung in kurzer Zeit</li></ul>	<ul style="list-style-type: none"><li>• Reimport notwendig zum Abfragen geschlussfolgerter Restriktionen</li><li>• keine Kennzeichnung des Pfades der SPARQL Abfrage</li><li>• keine dauerhafte Darstellung von Object Properties</li></ul>

# Toolchain

---

1

bisherige Situation

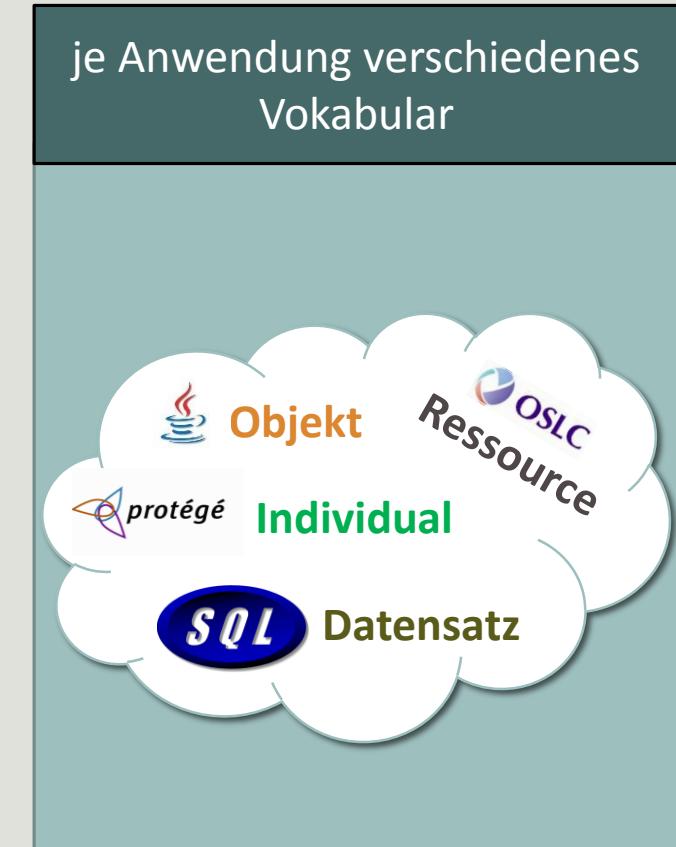
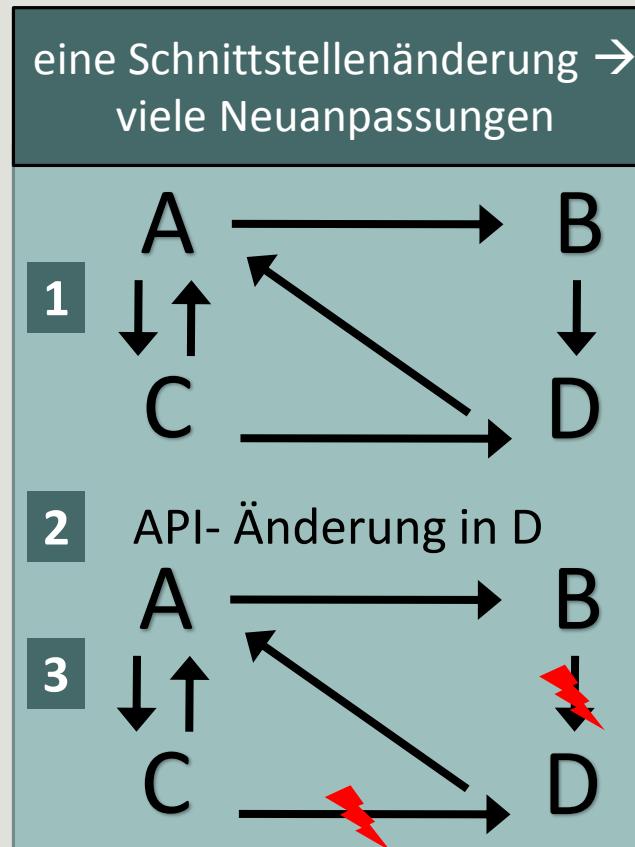
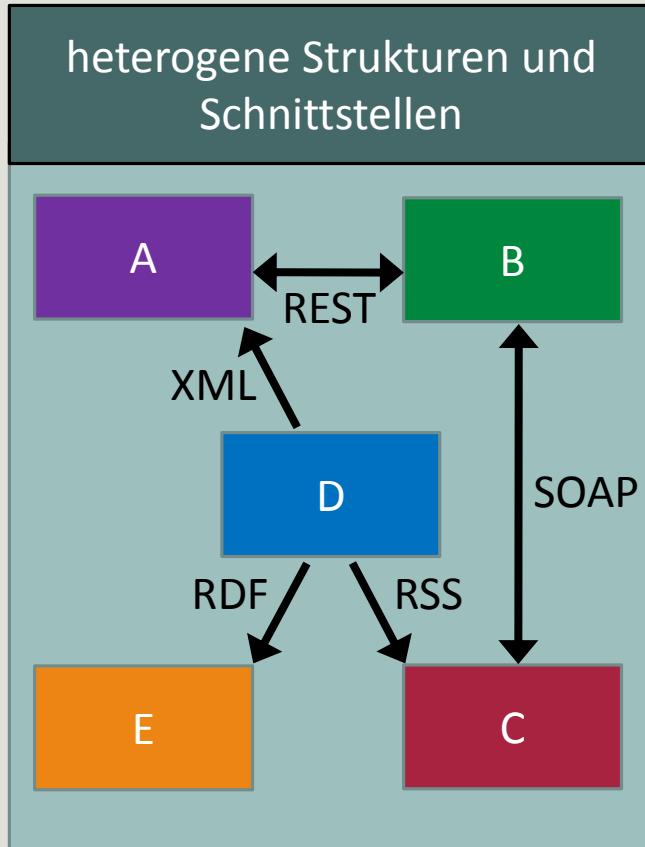
2

Representational State Transfer

3

Open Services for Lifecycle Collaboration

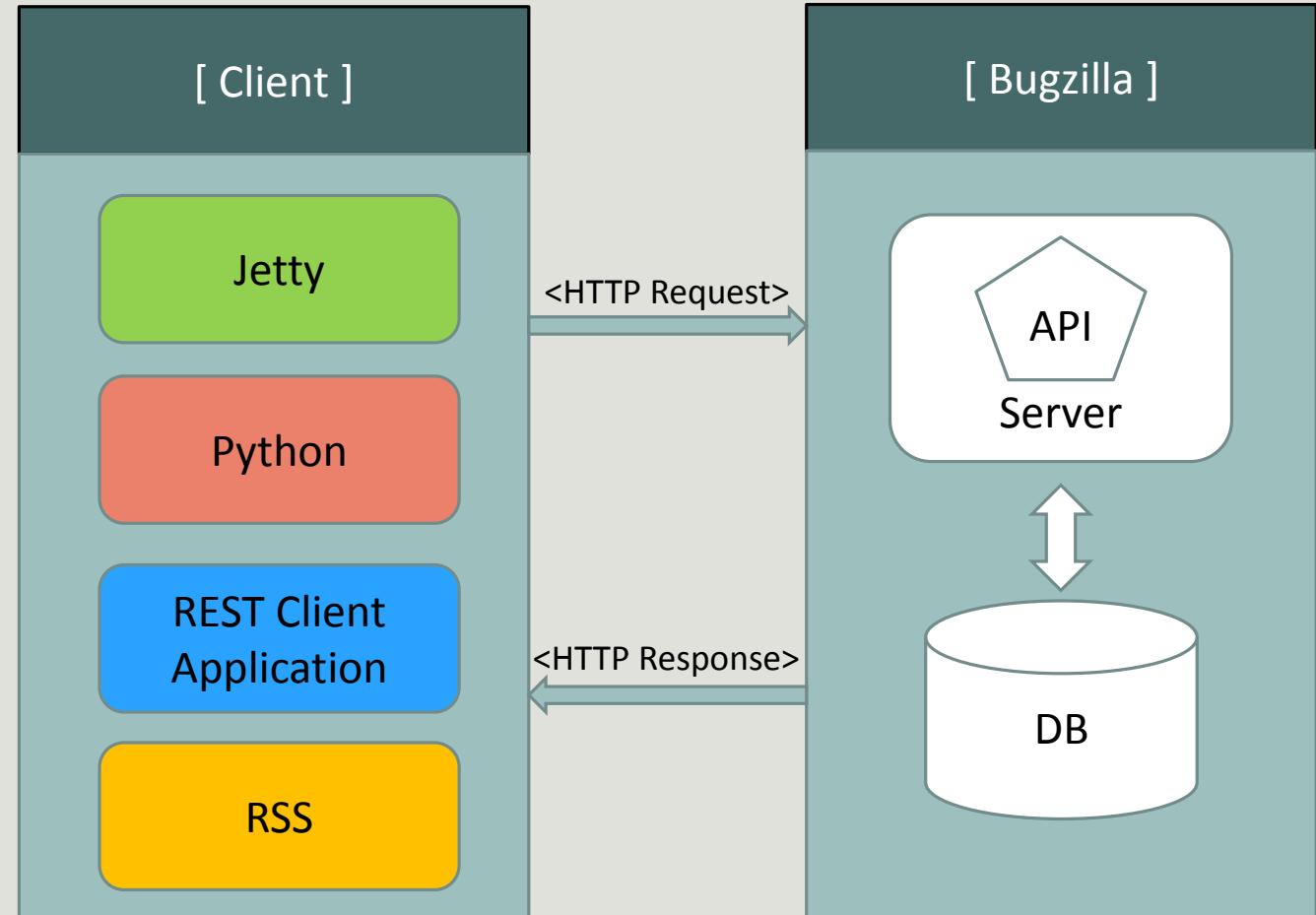
Bisher ist der Informationsaustausch zwischen verschiedenen Anwendungen sehr schwierig



# Representational State Transfer vereinheitlicht die Kommunikation

## Representational State Transfer

- jede Ressource eindeutig adressierbar, Namensräume wie in Ontologie
- unterschiedliche Repräsentanten (JSON, HTML, Atom) → Vielzahl an Clients
- HTTP-Basisoperationen: PUT, GET, POST, etc.



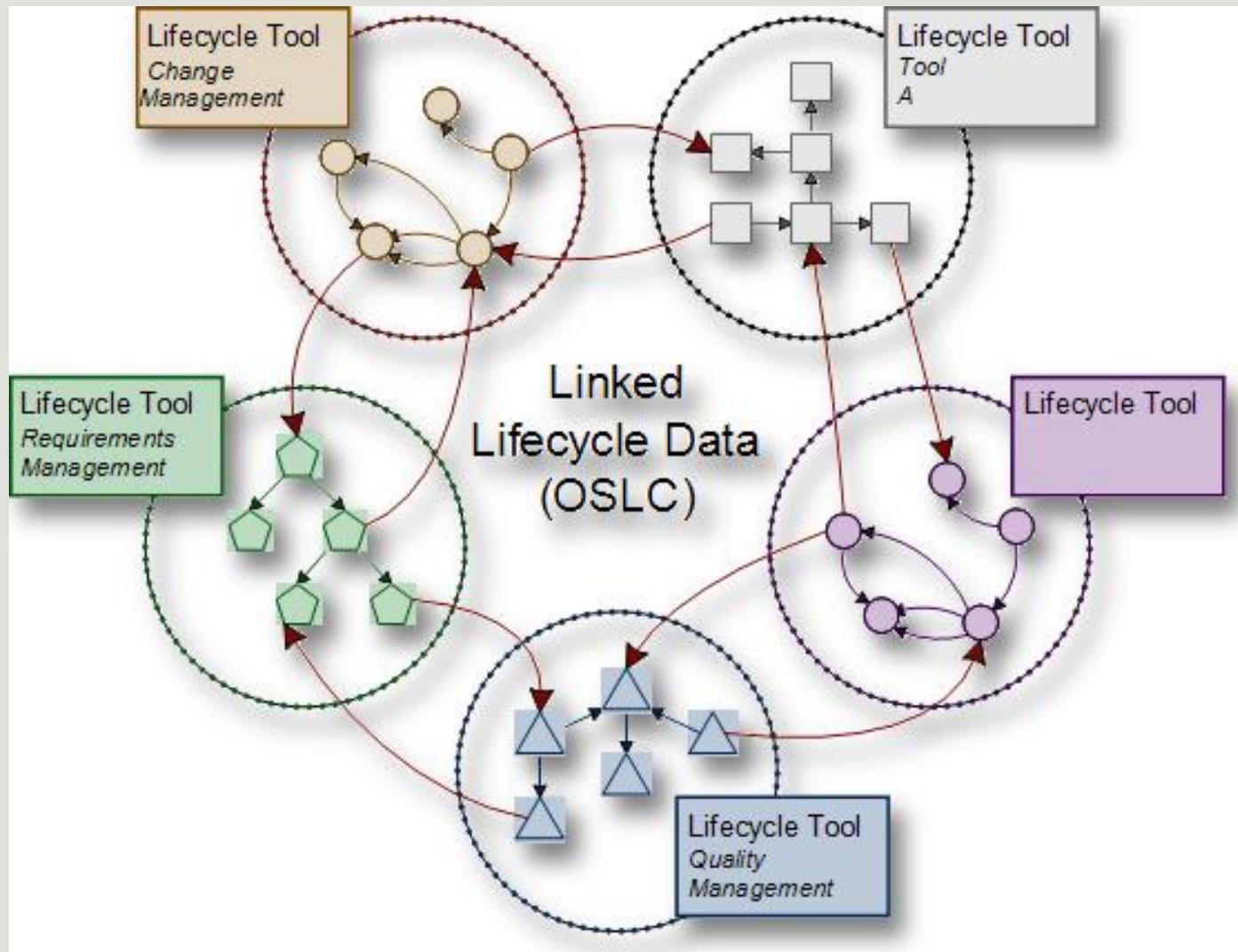
# Erleichterte Schnittstellenverwendung durch OSLC

---

## Open Services for Lifecycle Collaboration

- Aufwand für Verkettung verringern
- Vorbild Internet: jede Ressource wird von einem URL repräsentiert
- Integrationsansatz für Lebenszyklusmanagement von Software
- einheitliche Schnittstellen, einheitliches Vokabular
- CRUD: Create – Retrieve – Update - Delete

# Fünf typische Domänen des Lebenszyklusmanagements



[https://www.ibm.com/developerworks/mydeveloperworks/blogs/requirementsmanagement/resource/BLOGS\\_UPLOADED\\_IMAGES/oslc.png](https://www.ibm.com/developerworks/mydeveloperworks/blogs/requirementsmanagement/resource/BLOGS_UPLOADED_IMAGES/oslc.png)

(Lizenz:  
<http://creativecommons.org/licenses/by/3.0/us/>)

# Die Fortführung des Projektes ist zu empfehlen

---

Analyse und Integrationsplan der per REST anschließbaren Plattform

Schnittstellengenerierung aus SPARQL-Abfragen

Ausweiten des Ontologie-Vokabulars

Protégé um Funktionen und Restriktionen erweitern

# Backup-Folien

---

1

Agilität des Projektverlaufs und der Lösung

2

Anwendungen mit OSLC-Schnittstellen

3

Erweiterte SPARQL-Abfragen

4

OWLDoc als plattformunabhängige Dokumentation

# Agile Entwicklung und Lösung

---

- Agile Prinzipien
- Agile Methoden
- Agile Prozesse
- Agile Lösung

# Anwendung agiler Prinzipien bei der Durchführung des Projekts

---

- Frühzeitige und etwa zweiwöchentliche Telefonkonferenzen mit Bosch über aktuellen Entwicklungsstand
- Bereitstellen neuer Versionen der Ontologie ungefähr alle zwei Wochen
- Interne, persönliche Absprachen zum Projektstatus mindestens zweimal pro Woche
- Gemeinsamer Zugriff auf die gleichen Daten während der gesamten Projektlaufzeit
- Ständige Berichterstattung und Beurteilung des Fortschritts des Ontologie-Modells, der Software und Dokumentation
- Regelmäßiges Hinterfragen des Modelldesigns, des Software-Fortschritts und der Dokumentation
- Wahrung des Einfachheitsprinzips (KISS): Abstraktion geht über Spezifikation
- Selbstorganisation der Arbeitspakete in Kleingruppen

# Anwendung agiler Methoden und Prozesse bei der Durchführung des Projekts

---

## Agile Methoden:

Testgetriebene Entwicklung (Ontologie-Modell und SPARQL-Abfragen)

## Vorgehensrahmen agiler Prozess:

SCRUM:

1. Transparenz: regelmäßige, für alle Mitglieder sichtbare Berichterstattung über den Fortschritt
2. Überprüfung: Modell und Dokumentation regelmäßig geliefert sowie Produktzustand und Vorgehen beurteilt
3. Anpassung: situationsabhängige Verfahrensanpassungen in der Modellierung

# Wie agil ist die Lösung?

---

- Datenmodell kann in kleinen Schritten erweitert werden
  - SPARQL-Abfragen können als Modell für Schnittstellen in Java-Anwendung dienen und schrittweise aufgebaut werden
  - REST-Anbindung kann je Anwendung einzeln umgesetzt werden  
→ Schrittweises Umsetzen erlaubt schnellen Lerneffekt und Fortschritt
- 
- Neue Geschäftsbereiche können direkt hinzugefügt werden
  - Datenursprung kann leicht in andere Domäne verschoben werden  
→ Anpassbar an Veränderungen des Unternehmens

# Diese Anwendungen stellen OSLC-Schnittstellen bereit (Auswahl)

---

- IBM Jazz, Rational, Tivoli, SmartCloud
- HP Quality Center, Application Lifecycle Management
- Versionsverwaltung: Git
- Continuous Integration: Hudson, Jenkins
- Bugtracking: Bugzilla, Mantis Bug Tracker
- Microsoft Team Foundation Server, Sharepoint
- Salesforce

# Erweiterte SPARQL-Abfragen

---

- Alle Eigenschaften eines Individuals:

PREFIX ont: <...#>

SELECT ?s ?o

WHERE {<[http://..#Credit\\_Sparkasse](http://..#Credit_Sparkasse)> ?s ?o}

- Sortieren nach Spalte:

WHERE { .. } ORDER BY DESC(?s)

- Abfrage auf Menge limitieren:

WHERE { ... } LIMIT 10