Masterarbeit

Konzeptionierung eines service-basierten Backbones für durchgängiges Product Lifecycle Management

Thomas Psota

17.07.2018

Fachbereich Informatik

TU Kaiserslautern

Betreuer

Prof. Dr. Dr. h.c. H. Dieter Rombach

Dr. Martin Becker

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Kaiserslautern, 17.07.2018

Thomas Psota

Danksagung

Ich bedanke mich bei Martin Becker vom Fraunhofer IESE dafür, dass er sich bereit erklärt hat dieses Thema zu betreuen und für seine hilfreichen Ratschläge, ohne welche diese Arbeit in dieser Form nicht hätte existieren können.

Großer Dank gilt auch Prof. Martin Eigner vom Lehrstuhl für Virtuelle Produktentwicklung, welcher ausschlaggebend als Themensteller für diese Arbeit galt, und welcher mir in einem entscheidenden Moment in meinem Studium eine Perspektive geben konnte.

Mein größter Dank gilt meinem Opa Roman (*"Dziadek Romanek"*), meiner Oma Eugenia und meinem geliebten Hund Lucky, welche mich auf dem langen Weg, der mein Studium war, stets mit Beistand begleitet haben und die mich jetzt in diese nächste Etappe meines Lebens leider nicht mehr begleiten können.

Zusammenfassung

Der Produktentstehungsprozess unterliegt aufgrund neuer industrieller Trends, wie dem Internet of Things oder Industrie 4.0, einem großen Wandel. Unternehmen müssen darauf achten, dass ihre an der Produktentstehung beteiligten Anwendungen durchgängig miteinander vernetzt sind, um mit diesen sich stetig wandelnden Anforderungen in einem globalen Markt Schritt halten zu können. Momentane Integrationslösungen scheitern dabei noch durch zu starke Kopplungen an bestimmte Toolchains. In dieser Arbeit wurde die Integrationsplattform SPIDER entwickelt, mit welcher versucht wird diese Probleme zu lösen. Dabei wurde auf den Einsatz von Web-Technologien zurückgegriffen um starke Kopplungen der angebunden Systeme zu vermeiden. Zusätzlich lässt sich, mittels eines in dieser Arbeit entworfenen Generators, automatisch die Anbindung einer Datenquelle an SPIDER erzeugen. Durch die für SPIDER entwickelte Architektur wird dabei eine volle Abstraktion der angebundenen Systeme erreicht, so dass Änderungen der Datenquellen sich nicht negativ auf das Gesamtsystem auswirken.

Inhaltsverzeichnis

1 Einleitung

Der Wandel zu immer komplexer werdenden Produkten stellt Unternehmen vor neue Herausforderungen. Heutige Produkte bestehen schon lange nicht mehr aus reinen mechanischen Komponenten, sondern es werden immer mehr Mikrochips, Software und Sensoren eingebaut. Auch die Hinzunahme von Services und der Anbindung und Vernetzung von Geräten über das Internet of Things (IoT) spielt eine immer wichtigere Rolle. Man spricht bei diesen modernen Produkten auch von sog. *Smart Products*, also Produkten, welche über das Internet oder ein Netzwerk miteinander kommunizieren und über eingebaute Sensoren beispielsweise selbst in der Lage sind Diagnosen auszuführen, um früh Schäden zu entdecken und Ausfälle zu minimieren. [9][7]

Ein Umdenken der Unternehmen ist erforderlich, um auf die gestiegenen Anforderungen eines globalen Marktes reagieren zu können. Einer der grössten Zeit- und Kostenfaktoren ist hierbei die Produktentwicklung. Studien haben gezeigt, dass ein Großteil der Kostenverursachung eines Projektes in der Entwicklungsphase festgelegt wird, obwohl diese nur 10% der eigentlichen Kosten ausmacht. [9] Methoden und Prozesse für eine Unterstützung der frühen Entwicklungsphase und zur frühen Fehlererkennung, sind damit zur Pflicht für Unternehmen geworden, wenn diese ihre Wettbewerbsfähigkeit erhalten wollen. Dabei verlangt der steigende Anteil an Informatik, E-Technik und anderen Disziplinen, zusätzlich zur Mechanik, eine Umstellung auf eine inter-disziplinäre Produktentwicklung. [17]

Ebenso ist es durch die gestiegene Komplexität der Produkte für die meisten Unternehmen nicht mehr möglich diese modernen Produkte selbständig herzustellen, stattdessen wird für die Entwicklung und Produktion ein Netzwerk von Zulieferern aufgebaut, welche für jeweils einzelne Komponenten des fertigen Produktes zuständig sind. Ganze Industriezweige sind so entstanden, welche sich auf jeweils ein Teilgebiet spezialisieren. [17][16]

Auch die Anforderungen an die Produktion selbst haben sich verändert. Es existiert eine immer höhere Nachfrage an individualisierten Produkten. Man denke als Beispiel an die Hersteller von Automobilen, welche es dem Kunden erlauben bei der Bestellung eine Vielzahl an optionalen Features wie Sitzleder, Klimaanlagen, Karosseriefarben, usw. auszuwählen. Eine starre Massenproduktion ist damit nicht mehr möglich, stattdessen sind die Unternehmen auf ein übergreifendes Varianten Management angewiesen, sowie auf eine Vernetzung der Produktionsstätten, um auf individuelle Produktionsaufträge eingehen zu können. [9]

Aber nicht nur die Produkte selbst sind komplexer geworden, sondern auch die verschiedenen Prozesse die bei der Entwicklung, Produktion, Wartung und Entsorgung der Produkte anfallen haben sich in den letzten Jahren massiv verändert. Viele Unternehmen haben sich historisch bedingt bereits großflächige IT-Landschaften aufgebaut im ihre Prozesse zu unterstützen. Dabei fallen in den verschiedenen Phasen und Disziplinen eine große Menge an Daten von den verschiedenen IT-Systemen ab. Um mit dieser Flut an Daten, Dokumenten und Informationen zurecht zu kommen, hat sich das *Product Lifecycle Management* als Lösungsstrategie etabliert, um ein Produkt entlang seines gesamten Lebenszyklus mit dem Einsatz von speziellen IT-Lösungen zu unterstützen. [17]

Dabei findet ein reger Datenaustausch sowohl innerhalb der Unternehmen selbst wie auch mit den verschiedenen Zulieferern und Kunden statt. Ein global agierendes Unternehmen mit Entwicklungs- und Produktionsstätten auf der ganzen Welt muss beispielsweise Daten über Lagerbestände mit den Fabriken austauschen, sowie mit den Finanzierungsabteilungen und dabei darauf achten, dass diese Daten überall konsistent sind, ansonsten können Engpässe bei der Produktion auftreten, wenn leere Lager nicht durch rechtzeitige Bestellungen neu befüllt wurden. Ebenso müssen aktuelle Entwicklungsdaten sowohl zwischen den verschiedenen Abteilungen als auch mit den jeweiligen Zulieferern ausgetauscht werden, damit eine möglichst reibungslose Entwicklung stattfinden kann. [9] Dies kann sich als schwieriges Unterfangen gestalten, da es große Unterschiede in den eingesetzten Software-Lösungen als auch Datenmodellen der verschiedenen Unternehmen gibt. Sogar innerhalb eines Unternehmens selbst ist es nicht gewährleistet, dass alle Abteilungen bzw. Standorte die gleichen Softwaresysteme und Datenmodelle verwenden. [14]

Bisherige PLM-Lösungen sind noch gar nicht oder nur ansatzweise für diese globale und interdisziplinäre Produktentwicklung geeignet, weswegen es sich diese Arbeit zum Ziel setzt Möglichkeiten der Datenintegration für ein durchgängiges PLM zu finden. [9]

1.1 Problemstellung

Unternehmen verwenden heutzutage eine große Anzahl an verschiedenen IT Applikationen um ihre internen Prozesse und Abläufe zu unterstützen. Dabei sind großflächige IT-Landschaften entstanden, welche meistens unkontrolliert gewachsen sind. Dies ist darauf zurückzuführen, dass viele Abteilung innerhalb eines Unternehmens historisch bedingt ihre eigenen Tools und Applikationen eingeführt haben. Ein dabei oft beobachtetes Phänomen ist die Inselbildung innerhalb dieser Landschaften. Das heißt, eine isolierte Entwicklung sowie Abschottung der Tools untereinander. [14]

Dabei ist aufgrund der gestiegenen Anforderungen an den Produktentstehungsprozess (PEP) eine Integration zwischen immer mehr dieser Applikation erforderlich. Die Integration muss dabei über alle Dimensionen des PEP hinweg erfolgen. Das Schaubild in **Abb. 1** verdeutlicht die Ausmaße dieses Unterfangens. Es müssen also Applikationen im gesamten Lösungsraum des PEP miteinander vernetzt werden. Dies umfasst sowohl die einzelnen Domänen, als auch die einzelnen Phasen des Produktlebenszyklus sowie die gesamte Wertschöpfungskette mit ihren Standorten und Zulieferern.[9]

Eine durchgängige Integration aller dieser Anwendungen stößt dabei auf eine Reihe von Problemen. Zum Einen existieren große Unterschiede zwischen den einzelnen Applikationen selbst. Dies umfasst die Möglichkeiten der Datenabfrage oder die Datenformate. Für eine gelungene Integration müssen diese Unterschiede zwischen den Applikationen zunächst überbrückt werden. [14]

Aber auch wenn Unternehmen identische Anwendungen einsetzen, können Unterschiede entstehen, die eine Integration erschweren, da Unternehmen die eingesetzten Anwendungen oft anpassen oder erweitern lassen, um auf die individuellen Bedürfnisse des Unternehmens

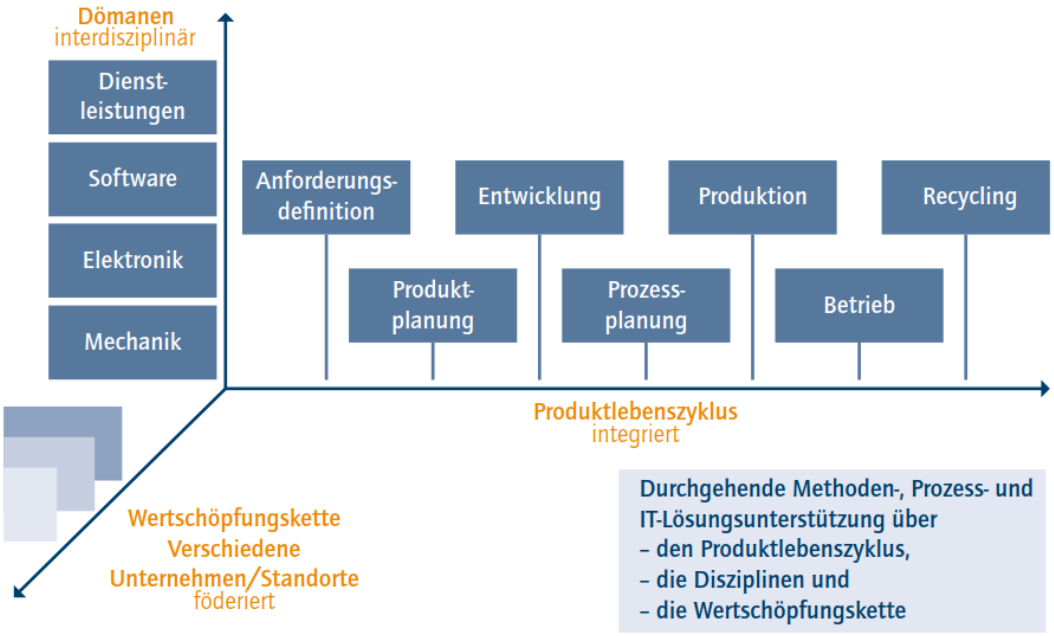

Abbildung 1: Dimensionen des modernen Produktentstehungsprozesses. Quelle: [9]

zugeschnitten zu sein. Aus diesen Gründen existieren auch keine fertigen *Out-of-the-box* PLM Lösungen. Jede PLM Lösung muss auf die individuellen Gegebenheiten des Unternehmens angepasst werden. Dies führt dazu, dass oft eine enge Kopplung an die integrierten Systeme erfolgt. [17]

Durch neue Trends, wie Smart Products oder das Internet of Things, unterliegen die IT-Landschaften eines Unternehmens aber weiterhin einem kontinuierlichen Wandel. So ist es entscheidend für ein Unternehmen, dass es auf Änderungen reagieren kann, und diese IT-Landschaften dementsprechend jederzeit verändern oder erweitern kann. Dabei stoßen die Unternehmen bei der Durchführung dieser Änderungen oft, durch die zu engen Kopplungen der integrierten Systeme, auf Probleme. [17]

Ein weiterer entscheidender Faktor ist der Datenaustausch mit Zulieferern oder Kunden sowie verschiedenen Standorten des Unternehmens. Oft wird von diesen ein direkter Zugriff an die PLM Lösung des Unternehmens benötigt. Auch hier kann es zu Problemen beim Datenaustausch kommen. Zum Einen existieren zu große Unterschiede bei den eingesetzten Schnittstellen um verlässlich auf diese Daten zuzugreifen. Zum Anderen gibt es auch Unterschiede bei den Datenformaten. Es existieren zwar Standards für Teilgebiete des PLM, wie bspw. der ISO Standard STEP AP 242 [36], aber es existieren noch keine übergreifenden Standards für PLM. [9]

1.2 Ziele der Arbeit

In dieser Arbeit sollen Möglichkeiten untersucht werden, um die im vorherigen Abschnitt vorgestellten Probleme lösen zu können. Dafür soll eine Integrationsplattform für die Integration von Datenquellen im PLM Umfeld erstellt werden. Im Kontext der Erstellung dieser Integrationsplattform beschäftigt diese Arbeit sich mit den folgenden Fragen:

- Welche Integrationsstrategien für PLM existieren bereits?
- Gibt es eine generische Architektur für eine Integration vorher unbekannter Datenquellen?
- Gibt es Möglichkeiten an die integrierten Daten zu gelangen, ohne explizites Wissen über die angebundenen System zu benötigen?
- Lässt sich eine lose Kopplung der angebundenen Systeme realisieren, damit Änderungen sich nicht auf die gesamte Plattform auswirken?

Die Integrationsplattform soll anhand eines konkreten Use Cases entwickelt werden. Dieser Use Case entstand aus einer Recherche gängiger Integrationsszenarien aus dem PLM Umfeld. Um ein Erfüllen der hier gesetzten Ziele zu messen, werden Qualitätskriterien für die Integrationsplattform festgelegt. Diese werden in einem späteren Kapitel vorgestellt.

1.3 Beiträge der Arbeit

Im Rahmen dieser Arbeit wurde SPIDER, eine generische Datenintegrationsplattform für die Integration verschiedener Datenquellen im PLM Umfeld, erarbeitet.

Dafür wurde zunächst aus einer Recherche über allgemeine Konzepte und Vorgänge des PLM ein Schlüsselszenario erstellt.

Dieses Schlüsselszenario diente als Basis für die Anforderungsanalyse und die Herleitung von Use Cases. Anhand dieser Use Cases wurde eine generische Architektur für die Datenintegrationsplattform entwickelt. Zusätzlich wurde aus der Betrachtung einer im PLM Umfeld eingesetzten Quellsysteme eine Ontologie für die semantische Beschreibung von Datenquellen angefertigt.

Die generische Architektur wurde innerhalb eines Prototypen implementiert, welcher anschließend in einer Machbarkeitsstudie validiert wurde. Dabei wurde, zusätzlich zur Integrationsplattform, ein Generator erstellt, welcher die für die Anbindung einer Datenquelle erforderlichen Wrapper erzeugen kann. Dafür wurde ein modellbasiertes Verfahren ausgehend von den semantischen Beschreibungen der Datenquelle als Input entwickelt.

Ein Übersichtsbild für die in dieser Arbeit unternommenen Schritte und die daraus erhaltenen Ergebnisse ist in **Abb. 2** zu sehen.

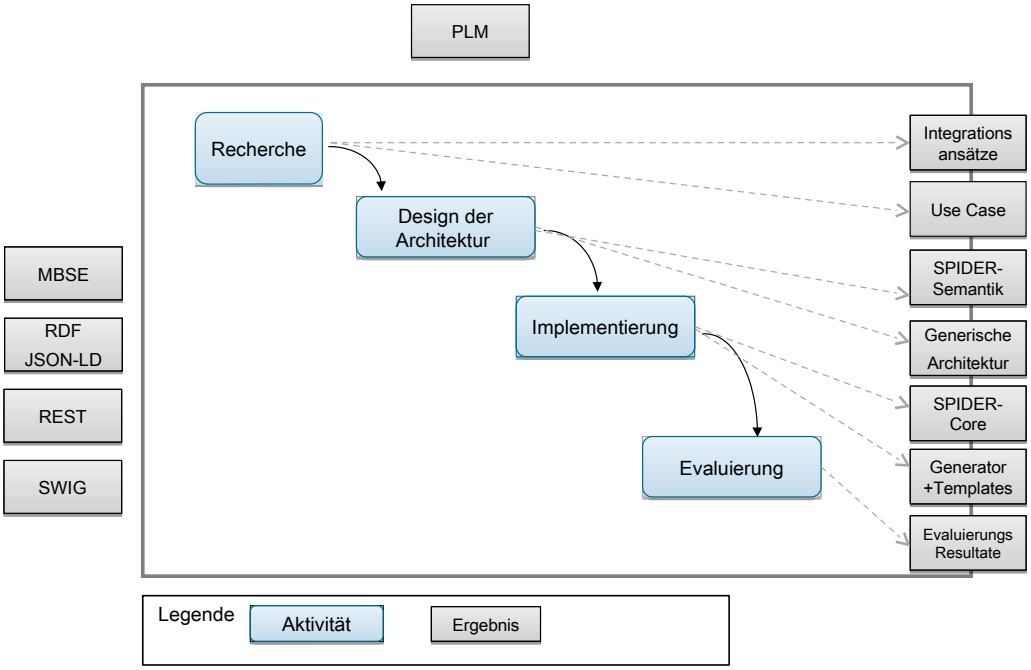

Abbildung 2: Übersichtsbild dieser Arbeit

1.4 Aufbau der Arbeit

Diese Arbeit ist unterteilt in sechs Kapitel.

Das erste Kapitel sollte einen kurzen Einblick in das Thema sowie über die generelle Struktur und Erzeugnisse der Arbeit geben.

In Kapitel 2 sollen zunächst die für das Verständnis dieser Arbeit benötigten Grundlagen vermittelt werden. Dabei werden grundlegende Konzepte des PLM sowie der Datenintegration vorgestellt. Am Ende des Kapitels findet ein Blick auf verwandte Arbeiten aus Forschung sowie Industrie statt.

Kapitel 3 ist das zentrale Kapitel dieser Arbeit. Dort wird das Konzept für die vorgeschlagene Datenintegrationsplattform aufgestellt und erörtert. Es werden zunächst die konkreten Anforderungen an die Architektur abgeleitet. Anschließend erfolgt eine Exploration verschiedener für die Konzeptionierung der Plattform verwendbarer Technologien.

In Kapitel 4 werden die bei der Implementierung der Integrationsplattform unternommenen Schritte erläutert. Dabei wird auf verwendete Technologien und Werkzeuge eingegangen, sowie auf die Erstellung der Core Assets, welche in dieser Arbeit verwendet wurden.

In Kapitel 5 findet eine Machbarkeitsstudie statt, in welcher das vorgeschlagene Framework an einigen im PLM Umfeld möglichen Datenintegrationsszenarien erprobt und evaluiert.

Die Ergebnisse der Arbeit werden in Kapitel 6 zusammengefasst. Anschließend werden mögliche Ausblicke und Verbesserungsansätze besprochen.

2 Grundlagen

Der Zweck dieses Kapitels ist es die benötigten Grundlagen zu vermitteln, auf denen der Rest der Arbeit aufsetzt.

Der erste Abschnitt bietet einen Einblick in die Grundlagen des PLM. Die hier vorgestellten Grundlagen dienen als Basis um die genaue Problemstellung und die Entscheidungsfindung dieser Arbeit besser nachvollziehen zu können. Da es sich bei PLM um ein sehr vielfältiges und vor allem großflächiges Thema handelt, werden nur die Teilgebiete näher betrachtet, welche für diese Arbeit relevant oder von besonderem Interesse sind.

Der zweite Abschnitt befasst sich mit allgemeinen Grundlagen und Konzepten der Datenintegration. Es werden die benötigten Definitionen und Konzepte vorgestellt und eine Übersicht über Techniken, Probleme und Architekturen der Datenintegration aufgestellt. Dieses Wissen wird benötigt, um die Entwicklung der Lösung in späteren Kapiteln nachvollziehen zu können.

Im dritten Abschnitt findet eine Betrachtung verwandter Arbeiten über Datenintegration im PLM Umfeld statt. Dabei werden zusätzlich zu Werken aus der Forschung auch Umsetzungen aus der Industrie angeschaut.

2.1 Product Lifecycle Management

Der Lebenslauf eines Produktes streckt sich von seiner Konzeption bis hin zur Produktion und letztlich zu seiner Außerbetriebnahme mehrere Phasen. [11]

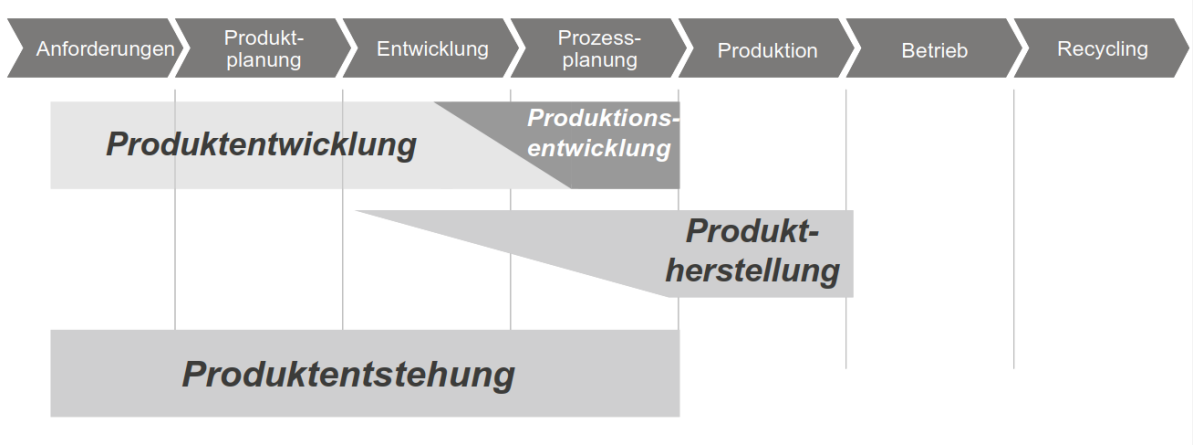

Abbildung 3: Phasen des Produktlebenszyklus. Quelle: [17, S.2]

Ein Überblick dieser Phasen ist in **Abb. 3** gegeben. Dort ist zu erkennen, dass der eigentliche Produktentstehungsprozess selbst nur einen Teil dieses Lebenszyklus ausmacht.

In der heutigen Zeit ist eine Produktentwicklung ohne den Einsatz von IT Lösungen undenkbar. Die *Virtuelle Produktentwicklung* (VPE) setzt es sich daher zum Ziel alle Phasen der Produktentwicklung durch den Einsatz von IT Lösungen zu unterstützen. Eigner et al definieren VPE wie folgt:

"[VPE] unterstützt alle Phasen der Produktentwicklung. Dies umfasst die frühe Phase von der Angebotsbearbeitung, über die Konzeption, die Entwicklung und Konstruktion, die Planung der Fertigungs- und Montageprozesse sowie das gemeinsame und umfassende Management aller auf das Produkt und die Produkti- onsplanung bezogenen Informationen in digitaler Form und deren Visualisierung. Alle Arbeitsprozesse und IT-Lösungen basieren darauf, die Entwicklungsstufen zu beschreiben, zu dokumentieren, zu optimieren, bzw. simulieren und die Informationen der nächsten Entwicklungsphase zur Verfügung zu stellen." [16]

Es geht dabei also vorrangig um die Erstellung von Daten.

Product Lifecycle Management (PLM) beschreibt ein Konzept für eine unternehmensweite Verwaltung aller Daten, welche während des Produktlebenszyklus erzeugt werden. Vereinfacht ausgedrückt, ist das Ziel des PLM die richtigen Informationen am richtigen Ort anzuzeigen, um ein effizienteres Arbeiten zu ermöglichen und dadurch Zeit und die damit verbundenen Kosten spart. [17, 11]

Der PLM Ansatz an sich besteht dabei sowohl aus konkreten IT Lösungen, als auch aus organisatorischen Strategien und muss individuell auf die Bedürfnisse eines Unternehmens zugeschnitten werden, denn je komplexer die Produkte und Prozesse des Unternehmens, sowie die dazugehörigen Daten und Dokumente, sind, desto komplexer gestaltet sich ein integriertes PLM. [11]

Im Folgenden soll eine kleine Einführung in die organisatorischen Konzepte und IT Lösungen gegeben werden.

2.1.1 Organisatorische Konzepte

Wie bereits erwähnt, besteht ein erfolgreiches PLM sowohl aus unterstützenden IT Lösungen, als auch aus organisatorischen Methoden, um die Daten und Dokumente, die in den einzelnen Phasen entstehen, zu verwalten. Im Folgenden sollen einige dieser organisatorischen Konzepte weiter beschrieben werden:

- **Nummernsysteme**

  Ein wichtiges Konzept von PLM ist es die Möglichkeit verwaltete Objekte (Dokumente, Artikel, etc.) durch einen Nummernschlüssel zu klassifizieren und zu identifizieren. [17]

  Die Vergabe von neuen Nummern wird durch das Nummernsystem festgelegt. Dabei ist gewährleistet, dass jede Nummer einzigartig ist. Ebenso ist eine Unveränderlichkeit einer bereits vergebenen Nummer garantiert. [17]

  Dies ermöglicht eine eindeutige Identifizierung eines Objektes. Die einzelnen Nummernsysteme sind dabei von Unternehmen zu Unternehmen unterschiedlich und meistens historisch entstanden, wie beispielsweise durch Übernahme der Nummerierung aus dem ERP System. Problematisch kann es werden, wenn mehrere Nummernsysteme in einem

Unternehmen gleichzeitig im Einsatz sind, wenn also möglicherweise ERP und PLM Systeme unterschiedliche Nummern verwenden. Eine eindeutige Identifizierung gleicher Artikel in beiden Systeme wäre dadurch erschwert. [17]

- **Dokumentenverwaltung**

  Da es sich bei der Produktentwicklung um eine klare Ingenieurstätigkeit handelt, ist die Erstellung von Dokumenten eine Unabdingbarkeit. Projektpläne werden aufgestellt, 2D-Zeichnungen und 3D-Modelle erstellt, Simulationsergebnisse erzeugt, technische Dokumentationen benötigt. An dieser Stelle sei der Einfachheit halber angenommen, dass alle Dokumente entweder direkt in digitaler Form erzeugt wurden, oder aber durch z.B. Einscannen digitalisiert wurden. Bei der Anzahl an erzeugten Dokumenten ist eine strikte Verwaltung erforderlich, um den Überblick behalten zu können. [11]

  Diese werden zunächst separat von den eigentlichen Produkten verwaltet. Die Beziehungen zwischen Produkt und Dokument werden anschließend festgehalten, um effizient feststellen zu können, welche Dokumente alle für ein Produkt relevant sind, und umgekehrt.

  Ebenso wie Produkte, erhalten auch Dokumente ihre eigenen Nummer, um eine eindeutige Identifizierung zu ermöglichen. Dabei existieren unterschiedliche Ansätze bei der Vergabe von Dokumentnummern. Diese können entweder komplett eigenständige Nummern sein, oder übernehmen die Nummer des jeweiligen Artikels, an welchem sie angefügt sind. [17]

  Für eine effektive Verwaltung von Dokumenten werden diese zusätzlich mit Metadaten gespeichert. Dies sind u.a. der Name des Dokuments, die Art des Dokuments (Zeichnung, techn. Dokumentation, etc.), Status/Reifegrad oder Änderungsindizes. [17]

- **Produktstrukturen**

  Ein Vorteil der virtuellen Produktentwicklung ist die Verwendung von Modellen. Mit diesen werden komplizierte Sachverhalten, wie der Zusammenbau eines komplexen Produktes auf die wesentlichen Details abstrahiert. Somit ist es möglich verschiedene Sichten auf einen gleichen Sachverhalt zu erhalten, welche auf unterschiedliche Anwendungsfälle zugeschnitten sind. [17]

  Eine der wichtigsten Sichten im PLM ist die Produktstruktur, oder auch Stückliste genannt. Während für einen Konstrukteur vor allem die exakte Positionierung eines Bauteiles in einer CAD-Zeichnung wichtig ist, beschränkt die Stückliste sich nur auf die Beziehungen zwischen den einzelnen Bauteilen eines Produktes und stellt diese hierarchisch dar. [17]

  Ein komplexes Produkt komplett nur durch Einzelteile zu beschreiben oder zu konstruieren, wäre nicht praktikabel. Stattdessen werden größere Produkte zunächst in kleinere

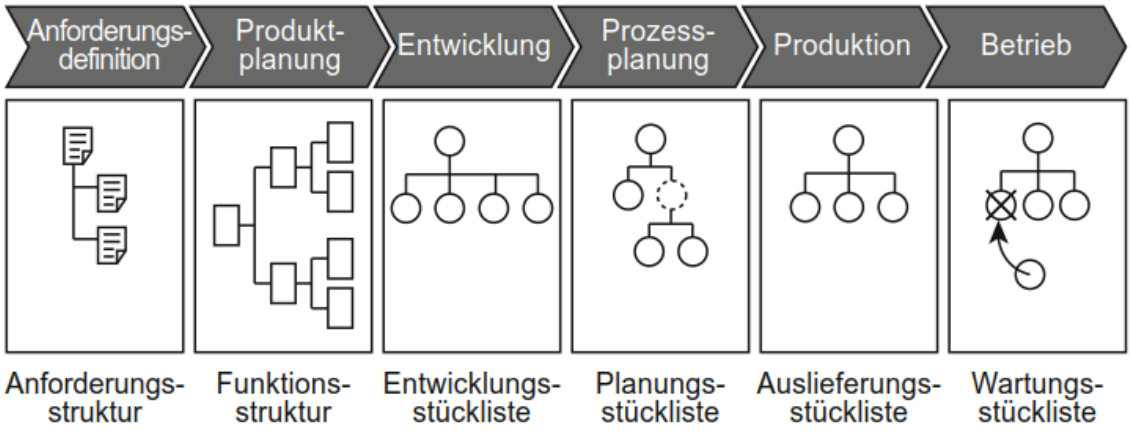

Abbildung 4: Produktstrukturen in einzelnen Produktlebensphasen. Quelle: [17, S. 79]

Gruppen zerlegt. Diese Baugruppen selbst, können ebenfalls wieder kleinere Baugruppen enthalten. Dieser Vorgang wird so tief durchgeführt, bis eine Baugruppe nur noch aus atomaren Einzelteilen besteht. [17]

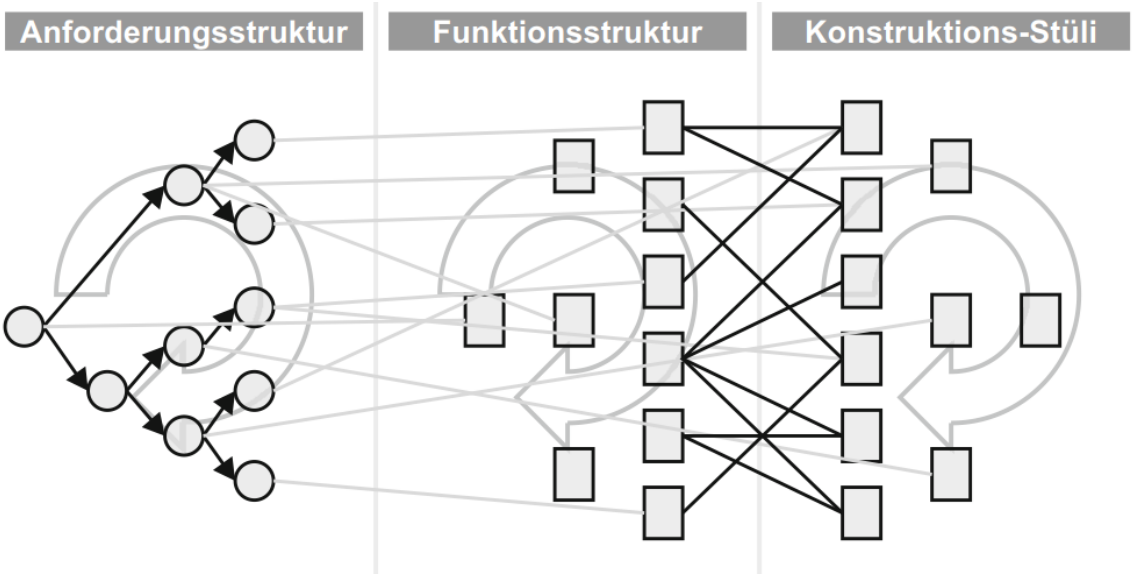

Abbildung 5: Verknüpfung der Anforderungs- und Funktionsstruktur mit der Entwicklungsstückliste. Quelle: [17, S.79]

Eine beliebte Darstellungsform der Stücklisten ist die Ansicht als Graph. In diesem ist schnell sichtbar in welchen Baugruppen ein Teil verbaut ist, oder aus welchen Teilen sich eine Baugruppe zusammensetzt. Die Stückliste wird dabei von oben nach unten betrachtet ("*top-down*"). In umgekehrter Richtung ("*bottom-up*"), ausgehend von einer Komponente, erhält man eine Ansicht über alle Produkte in denen diese Komponente verbaut ist. [17]

Erweitert man das Strukturmodell durch Hinzunahme von allen produktbeschreibenden Daten, erhält man das integrierte Strukturmodell. In diesem werden zu jedem Bauteil noch weitere Informationen angefügt, wie beispielsweise CAD oder andere Dokumente.

[11]

Die Struktur der Stückliste ist dabei längst nicht eindeutig und abhängig von der Phase in der sie hauptsächlich benötigt wird. So spricht man in der Entwicklungsphase auch von der Entwicklungsstückliste, (auf englisch auch E-BOM (Engineering Bill-of-material) genannt) welche hauptsächlich die Struktur des Produktes wie es entwickelt wurde wiederspiegelt, wird in der Produktion auf die Fertigungsstückliste zurückgegriffen wird. Weitere Arten von Produktstrukturen ist in **Abb. 4** gegeben. [17]

Von besonderem Interesse für Industrie und Forschung ist die Verknüpfung zwischen diesen verschiedenen Produktstrukturen, wie sie beispielsweise in **Abb. 5** zu sehen ist. [17]

Da die verschiedenen Produktstrukturen und Listen in ihren eigenen, spezialisierten Systemen verwaltet und gespeichert werden, werden übergreifende Integrationsstrategien benötigt, um diese Problemstellung zu lösen. Dies ist vorrangig eines der Probleme, welche diese Arbeit versucht anzugehen.

2.1.2 Technische Konzepte

Sind die organisatorischen Voraussetzung für eine Einführung von PLM erfüllt, werden IT gestützte Tools benötigt, um bei der Umsetzung dieser Maßnahmen zu helfen.

Für die eigentliche Entwicklung setzen Unternehmen seit Jahren vermehrt auf den Einsatz von spezialisierten IT-Anwendungen, wie etwa den CAD-Tools für die Konstruktion. Eine PLM Lösung muss demnach in der Lage sein, die anfallenden Daten aus der gesamten IT-Landschaft verwalten zu können. Sie bildet somit das Backbone der Datenverwaltung eines Unternehmens.

In diesem Abschnitt sollen einige dieser Systeme und ihre Hauptfunktionen vorgestellt werden.

- **Produkt Daten Verwaltung**

  Historisch bedingt setzt die Funktionalität von PLM Lösungen auf den *Produkt Daten Management*-Systemen (PDM) auf, welche sich nur auf den Bereich der Produktentstehung beschränkten. Zusätzlich zu den im vorherigen Kapitel beschriebenen Funktionen übernehmen PLM Lösungen damit auch gleichzeitig die Funktionen der PDM Systeme. [17]

  Dazu zählen die Verwaltung von Stammdaten, Produktstrukturen, Prozessstrukturen, die Verwaltung der dazugehörigen Dokumente. Zusätzlich kümmert sich ein PDM System um die Versionierung aller verwalteten Daten. [17]

  Die Verwaltung von Stammdaten beschränkt sich dabei hauptsächlich auf die Verwaltung von Metadaten der einzelnen Artikel oder Dokumente. Darunter fallen Teilenummern, Herstellungskosten, Informationen über den Autor, Erstellungsdatum, etc.

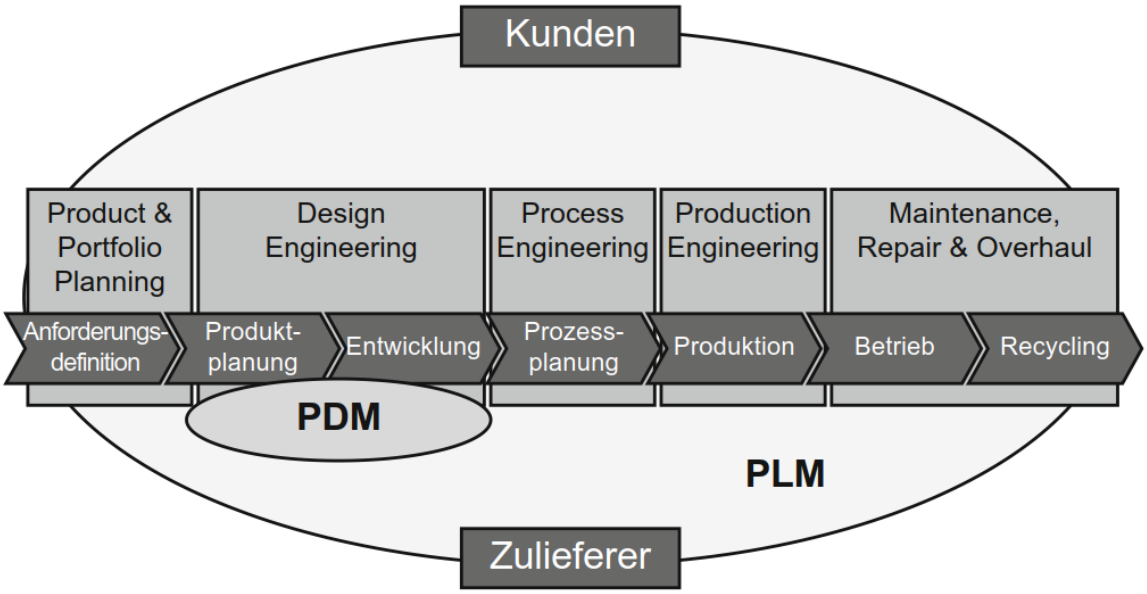

Abbildung 6: Aufgabengebiete von PDM und PLM im Produktlebenszyklus. Quelle: [16, S.270]

Diese Funktionalitäten werden dann durch die Kernfunktionen von PLM erweitert, d.h. eine Erweiterung des PDM Ansatzes für alle weiteren Phasen des Lebenszyklus sowie alle beteiligten Abteilungen und Disziplinen. In **Abb. 6** wird noch einmal die Abgrenzung zwischen PDM und PLM verdeutlicht. [17]

- **Autorensysteme**

  Historisch bedingt besitzen die meisten Unternehmen bereits vor der Einführung von PLM großflächige IT-Landschaften, welche über die Jahre aufgebaut wurden und gewachsen sind.

  Dabei gibt es je nach Art des Unternehmens und der Branche unzählige verschiedene Software-Systeme, welche zum Einsatz kommen können.

  Die grundlegenden produkt-relevanten Daten selbst werden in sog. *Autorensystemen* erzeugt. Computer Aided Design (CAD)-Systeme machen den Großteil dieser Systeme aus. [17] Diese existieren für die Anwendungsbereiche:

  - Mechanik (M-CAD)
  - Elektrotechnik und Elektronik (E-CAD)
  - Softwareentwicklungssysteme (CASE)

- **Customizing**

  Viele Unternehmen bauen sich über die Jahre ihre eigenen Datenmodelle, basierend auf den verwendeten IT-Applikationen und den firmen-internen Bedürfnissen, auf. Selbst wenn zwei Firmen komplett identische Applikationen verwenden, kann es zu großen Unterschieden innerhalb der Datenmodelle kommen. [14]

Deswegen sind kommerzielle PLM Lösungen nicht Öut-of-the-Boxëinsetzbar, sondern müssen erst auf das individuelle Datenmodell und die verwendeten Applikationen eines Unternehmens angepasst werden. Man spricht bei diesem Vorgang von *Customizing*. [17]

Dabei werden von den Entwicklern der Systeme meistens spezielle Tools um die Software anzupassen zur Verfügung gestellt. Alternativ nehmen die Entwickler auch individuelle Änderungswünsche eines Unternehmens in Auftrag und liefern eine Variante der Software aus. [17]

Änderungen werden dabei oft am Datenmodell, der Oberflächengestaltung sowie am Verhalten der Systeme vorgenommen. [17]

2.1.3 Model Based Systems Engineering

Der klassische PLM Ansatz im Systems Engineering ist von papier- oder dokumentbasierten Abläufen geprägt. Model Based Systems Engineering (MBSE) ist ein modellbasiertes Vorgehensmodell und stellt eine Weiterführung des traditionellen PLM Ansatzes dar.

Im Mittelpunkt von MBSE steht die Erstellung von computergestützten Systemmodellen in jeder Phase des Lebenszyklus eines Produktes mit dem Ziel diese in die Nächste Phase des Lebenszyklus weiterzugeben, sowie für Validierung und Simulation zu verwenden. [15]

Zur Erstellung der Modelle wird die Modellierungssprache SysML verwendet. Bei SysML handelt es sich um eine Erweiterung von UML mit der Zielsetzung Aktivitäten des Systems Engineering zu unterstützen. [31]

Diese Systemmodelle sollen dabei in der frühen Entwicklungsphase zum Einsatz kommen, und durch qualitative Modelle die Spezifikationen eines Produktes beschreiben. Da die erzeugten Modelle damit zu einem wichtigen Bestandteil des Entwicklungsprozesses werden, ist eine Integration zwischen den SysML-Autorenwerkzeugen und PLM Systemen zwingend notwendig. [15]

Im Rahmen des MBSE Forschungsprojektes mecPro^2 wurde zusätzlich die Verwaltung semantischer Netze als neu benötigte PLM Funktionalität identifiziert. Traditionell werden im PLM System die verschiedenen Strukturen isoliert gespeichert und betrachtet. Dies sind typischerweise die Anforderungstrukturen, Funktionsstrukturen und Stücklisten. Es besteht die Notwendigkeit, dass ebenfalls die Verbindungen zwischen diesen Strukturen im PLM System verwaltet werden. Ein Beispiel für eine mögliche Vernetzung dieser Strukturen ist in **Abb. 7** gezeigt. [15]

Bis heute mangelt es dabei an einer Methodik zur Integration bzw. zur Gestaltung bzw. Modellierung der Schnittstelle in den Modellen. Dies erfordert neue Ansätze für die Speicherung und Verwaltung dieser netzartigen Strukturen und stellt heutige PLM Systeme vor eine große Herausforderung. [15]

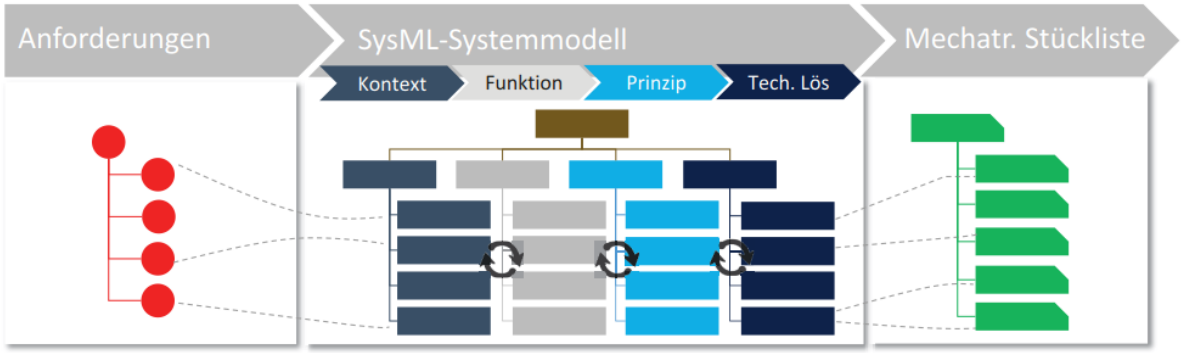

Abbildung 7: Beispiel für Vernetzungen zwischen Systemmodellen in MBSE Quelle: [15]

2.2 Datenintegration

2.2.1 Verteilung

Das erste Problem bei der Integration von Daten ergibt sich, wenn diese in verschiedenen Systemen liegen. Man spricht von Verteilung. Diese hat zwei Aspekte, die physische und die logische Verteilung. [24]

Eine physische Verteilung liegt vor, wenn die Daten sich auf physisch getrennten Systemen befinden. Oft sind diese dann auch geografisch entfernt. Um auf die Daten in diesen Systemen zugreifen zu können, müssen diese zunächst in einem Netzwerk auffindbar und ansprechbar sein. Mittlerweile existieren Techniken, um diesen Vorgang für Anwendungen sowohl im Internet als auch im Intranet transparent zu halten. Dadurch ergeben sich spezielle Anforderungen an die Performance der Integrationssysteme, da jeder Datenzugriff in ein verteiltes System nun der Zugriffslatenz des Netzwerks ausgesetzt ist. [24]

Von logischer Verteilung wird gesprochen, wenn Daten (auch im selben System) sich an verschiedenen Stellen befinden. Ein Beispiel aus der PLM wäre das Vorhandensein eines Bauteils im PDM und im ERP System. Man spricht dabei von Redundanz im System. Dabei unterscheidet man zwischen *kontrollierter* und *unkontrollierter* Redundanz. Der erste Fall wäre beispielsweise bewusste die Verteilung und Duplizierung von Datenbeständen zur Vermeidung von Datenausfällen. Unkontrollierte Redundanz hingegen entsteht durch historische gewachsene oder organisatorisch bedingte Verteilung von Daten. [24]

2.2.2 Autonomie

Autonomie im Kontext der Datenintegration beschreibt die Freiheit der Datenquellen unabhängig über die von ihnen verwalteten Daten, ihre Strukturen und ihre Zugriffsmöglichkeitenn, zu entscheiden. Dies ist fast immer der Fall, wenn bereits vorhandene Systeme integriert werden sollen.

Autonomie erzeugt fast immer schwierige Heterogenitätsprobleme. Zwar ließen sich diese Probleme umgehen, indem man die Autonomie der eingesetzten Systeme einschränkt, in der

Realität ist dies aber nur sehr selten möglich, weswegen man die dadurch entstehenden Folgen bei der Datenintegration berücksichtigen muss. [24]

Autonomie lässt sich hauptsächlich in vier Arten unterscheiden:

- **Designautonomie:**

  Diese beschreibt die Freiheit einer Datenquelle frei darüber zu unterschieden, wie sie ihre Daten zur Verfügung stellt. Darunter fallen:

  - Datenformate
  - Datenmodelle
  - Schemata
  - Syntaktische Darstellung der Daten
  - Verwendung von Schlüssel- Begriffsystemen
  - Einheiten von Werten

  Designautonomie ist oft die Ursache für das Auftreten von Schwierigkeiten bei der Integratrion. Eine Möglichkeit diese Probleme zu umgehen, ist der Einsatz von Standards für den Datenaustausch, wie es beispielsweise in der Automobilindustrie der Fall ist. [24]

- **Schnittstellenautonomie:**

  Hierunter fällt die Freiheit einer Datenquelle, selbst zu bestimmen, welche Schnittstellen sie für den Datenzugriff anbietet. Dies umfasst u.a. Protokolle und Anfrageformate für den Datenaustausch. Die Schnittstellenautonomie hängt stark mit der Designautonomie zusammen, da die Art des Zugriffs oft von der Datenrepräsentation abhängt oder zumindest stark eingeschränkt wird. [24]

- **Zugriffsautonomie:**

  Zugriffsautonomie liegt vor, wenn eine Datenquelle selbst entscheiden kann, wer auf ihre Daten zugreifen darf. Dies umfasst Fragen der Benutzerkonten, Authentifizierung und Autorisierung. Darunter fallen auch Lese- und Schreibrechte, sowohl für die gesamte Datenquelle, als auch fein geregelte Zugriffskontrolle auf einzelne Datensätze. [24]

- **Juristische Autonomie:**

  Juristische Autonomie beschreibt das Recht einer Datenquelle, die Integration ihrer Daten in ein Integrationssystem zu verbieten, wenn es dadurch beispielsweise zu Urheberrechtsverletzungen kommen könnte. Da im Kontext dieser Arbeit davon ausgegangen wird, dass die zu integrierenden Datenquellen sich alle unternehmensintern befinden und es auch sonst keine weiteren juristischen Bedenken gibt, soll an dieser Stelle nicht weiter auf diese Art der Autonomie eingegangen werden. [24]

Zusätzlich zu den oben genannten Formen der Autonomie, existieren auch noch die Freiheiten einer Datenquelle jederzeit ihre Zugriffsrechte, Schnittstellen oder Datenformate ändern zu können. Dieses als *Quellevolution* bezeichnete Phänomen stellt mögliche Integrationssysteme vor die Herausforderungen auf diese Änderungen zu reagieren. Ein wichtiges Qualitätskriterium für Integrationssysteme ist deswegen die Möglichkeit schnell und flexibel auf Änderungen der unterliegenden Quellsysteme zu reagieren. Bei Integrationsszenarien innerhalb eines Unternehmens lassen sich üblicherweise Einschränkungen bei der Autonomie der einzelnen Quellsysteme vornehmen. So sind beispielsweise die Datenformate und Schnittstellen der bereits eingesetzten Quellsysteme im voraus bekannt. Andererseits können die Integrationsteams auch vorab über anstehende Änderungen informiert werden bzw. Vorgaben über selbiges machen. Diese Änderungen laufen dann über im Unternehmen festgelegte Prozesse ab. [24]

2.2.3 Heterogenität

Leser und Naumann definieren die Heterogenität zweier Systeme als: *"Zwei Informationssysteme, die nicht die exakt gleichen Methoden, Modelle und Strukturen zum Zugriff auf ihre Daten anbieteten, [...]."* [24, S.58]

Heterogenität steht konzeptuell zunächst orthogonal zur Verteilung zweier Systeme. So können zwei Datenquelle physikalisch verteilt sein, und sind dennoch homogen. In gleicher Weise können sich zwei Datenquellen auf dem selben Rechner befinden, und dennoch komplett heterogen sein. Es ist zu beobachten, dass die Heterogenität zweier Systeme mit der Autonomie der Quellen gekoppelt ist. In der Praxis sind damit zwei vollkommen unabhängige Datenquellen fast immer heterogen, auch wenn diese die selben Informationen verwalten. [24]

Heterogenität entsteht dabei aufgrund unterschiedlicher Anforderungen, unterschiedlicher Entwickler und unterschiedlicher zeitlicher Entwicklungen. Auch zunächst identische Softwaresysteme, können heterogen sein, da diese in der Praxis an die unterschiedlichen Anforderungen von Unternehmen angepasst werden. Man spricht hierbei von *Customizing*. [24]

Heterogenität lässt sich grob in folgende Kategorien aufteilen:

- **Technische Heterogenität:**

  Als technische Heterogenität bezeichnet man Unterschiede zwischen Datenquellen, die sich nicht unmittelbar auf Daten und ihre Beschreibung beziehen, sondern auf die Möglichkeit des Datenzugriffs. Eine Übersicht der verschiedenen Ebenen der technischen Heterogenität ist in **Tabelle 1** gegeben. [24]

  Zunächst wird unterschieden, welche Möglichkeiten ein System zur Verfügung stellt, um dessen Daten anzufragen. Dies kann über Funktionen, Anfragesprachen, o.ä. passieren. Auf nächster Ebene wird unterschieden, in welcher Ausprägung die Anfragemöglichkeit vorliegt, also ob beispielsweise SQL oder XQuery als Anfragesprache verwendet wird. Gestellte Anfragen werden dann in einem bestimmten Format beantwortet. Dies können beispielsweise proprietäre Binärdaten sein, oder es kann auf standardisierte Formate zurückgegriffen werden. [24]

Ebene	Mögliche Ausprägungen
Anfragemöglichkeit	Anfragesprache, parametrisierte Funktionen, Formulare (engl. canned queries)
Anfragesprache	SQL, XQuery, Volltextsuche
Austauschformat	Binärdaten, XML, HTML, tabellarisch
Kommunikationsprotokoll	HTTP, JDBC, SOAP

Tabelle 1: Technische Ebenen der Kommunikation zwischen Integrationssystem und Datenquellen. Quelle: [24, S.62]

- **Syntaktische Heterogenität:**

  Syntaktische Heterogenität bezeichnet Unterschiede in der Darstellung gleicher Sachverhalte. Darunter fallen technische Aspekte wie beispielsweise unterschiedliche Zeichenkodierungen bei Textformaten (ASCII, Unicode). Auch die Darstellung von Zeit- und Datumsformaten fällt darunter. Syntaktische Heterogenität stellt im Kontext der Datenintegration kein Problem dar, da sich die verschiedenen Formate meistens leicht konvertieren lassen. [24]

- **Datenmodell Heterogenität:**

  Datenmodell Heterogenität liegt vor, wenn Unterschiede in den Datenmodellen zwischen Datenquellen existieren. Wenn also beispielsweise Daten in einer Datenquelle in Form von SQL Tabellen gespeichert werden, und in einer anderen als XML Dokumente.

  Dies umfasst noch nicht die Problematik der semantischen Überlappung von Daten aus verschiedenen Datenquellen, ist aber dennoch stark damit gekoppelt, da Heterogenität im Datenmodell fast immer semantische Heterogenität hervorruft. [24]

- **Strukturelle Heterogenität:**

  Auch wenn bei der Modellierung eines Sachverhalts aus der Realwelt das gleiche Datenmodell verwendet wird, kann es dennoch Unterschiede in den beiden entstandenen Schematas geben. Dies ist auf die hohen Freiheitsgrade bei der Übersetzung von konzeptionellen in logische Modelle. Weitere Ursachen lassen sich auch auf verschiedene Anforderungen an das Schema zurückführen, wie beispielsweise Optimierungen aufgrund von Performance. [24]

- **Schematische Heterogenität:**

  Schematische Heterogenität ist ein Spezialfall der strukturellen Heterogenität und liegt vor wenn unterschiedliche Elemente eines Datenmodells verwendet werden, um denselben Sachverhalt zu modellieren. So können beispielsweise Sachverhalte in relationalen Modellen jewils als Relation, als Attribut oder als Wert modelliert werden, und sind dabei dennoch semantisch gleichwertig. Ein häufiges Problem der schematischen Heterogenität ist, dass schematische Konflikte sich nicht durch Mittel von Anfragesprachen überbrücken lassen. So müssen in relationalen Sprachen alle Attribute und Relationen explizit benannt werden. Bei Änderungen an diesen Elementen, muss dementsprechend auch die Anfrage angepasst werden. [24]

- **Semantische Heterogenität:**

  Eine der grössten Herausforderungen bei der Datenintegration ist die semantische Heterogenität. Daten werden für einen menschlichen Benutzer erst durch Interpretation zu Information. Dafür wird Wissen über die konkrete Anwendung, als auch das relevante Domänenwissen benötigt. So werden zur Interpretation von Daten u.a. weitere Informationen aus der Datenquelle herangezogen:

  - Der Name des Schemaelements
  - Die Position des Schemaelements im gesamten Schema
  - Wissen über die konkrete Domäne

  Diese Informationen werden unter dem Begriff *Kontext* zusammengefasst. Dieser kann in computerlesbarer Form vorliegen, wie beispielsweise das Schema im Datenmodell, oder aber komplett nicht im Datenmodell abgebildet worden sein, und damit für den Programmierer nicht erreichbar. [24]

2.2.4 Transparenz

Das Hauptziel der Datenintegration ist es die verschiedenen auftretenden Heterogenitäten in den Datenquellen vor dem Benutzer zu verstecken. Man spricht dabei von *Transparenz*. [24]

Diese kann in verschiedenen Ausprägungen auftreten:

- **Ortstransparenz:**

  Hier soll dem Benutzer oder der Anwendung verborgen werden, an welchem physischen Ort sich das Quellsystem befindet. Dabei muss es so erscheinen, als ob die Daten lokal vorhanden wären, der Benutzer also keinerlei Wissen über die Erreichbarkeit der Datenquelle im Netzwerk benötigt. [24]

- **Verteilungs- bzw. Quellentransparenz:**

  Hier soll verborgen werden, dass sich die Daten in verschiedenen Quellsystemen befinden. Kenntnisse über die jeweils eingesetzten Schemata der Quellsysteme bleiben dem Benutzer verborgen. Auch hat der Benutzer keine Kenntnis über die Herkunft der Ergebnisse. Betrachtet von außerhalb, verhält sich das Integrationssystem wie eine zentrale, homogene Datenbank.

  Oft ist eine komplette Verteilungstransparenz nicht erwünscht, da der Benutzer sehr wohl wissen möchte, aus welchem Quellsystem eine bestimmte Information stammt. [24]

- **Schnittstellentransparenz:**

  Diese Art der Transparenz verbirgt dem Benutzer welche Methoden und Schnittstellen verwendet wurden, um die Datenquelle anzusprechen. [24]

- **Schematransparenz:**

Bei dieser Art der Transparenz werden die strukturellen Heterogenitäten zwischen den Datenquellen und dem Integrationssystem verborgen. Benutzer brauchen dementsprechend kein Wissen über die einzelnen Datenmodelle der Datenquellen, und arbeiten nur auf dem globalen Schema des Integrationssystems. Das Integrationssystem muss dabei Anfragen auf das globale Schema in die jeweiligen Schemata der Datenquellen übersetzen. [24]

2.2.5 Architekturen

Es gibt zwei generelle Ansätze der Datenintegration: die *materialisierte* und die *virtuelle* Integration. Der Hauptunterschied beider Ansätze, ist der Speicherort der zu integrierenden Daten.

- **Materialisierte Integration:**

  Bei der materialisierten Integration, werden Kopien der Daten aus den Quellsystemen gezogen, und in einem zentralen Speicherort in einem globalen Schema gespeichert. Dies passiert nach dem *Push-Prinzip*; es werden also periodisch Updates der Quellsysteme in die globale Datenbank eingespielt, meist zu festgelegten, wie beispielsweise einmal täglich. Nach der Übertragung findet ein Datenreinigungs-Prozess statt, welcher die Daten normalisiert, Duplikate entfernt und Fehler beseitigt. [24]

  Anfragen über diese Daten werden dann an die zentrale Datenbank gestellt und auch komplett von dieser verarbeitet. Die Quellsysteme sind hierbei nicht mehr involviert, was unter anderem einen erheblichen Geschwindigkeitsvorteil mit sich bringt. [24]

  Ein Nachteil von materialisierten Integrationssystemen, ist die Tatsache, dass das System nicht immer die aktuellsten Daten zur Verfügung hat. Auch kann es schnell zu Konsistenzproblemen kommen, wenn Änderungen an den Datenbeständen vorgenommen werden. [24]

  Eine weitverbreitete Form materialisierter Integrationssysteme sind die Data Warehouses. Diese werden häufig für Datenanalysezwecke verwendet.

- **Virtuelle Integration**

  Im Gegensatz zur materialisierten Integration steht die virtuelle Integration. Bei dieser bleiben die Daten gänzlich in den Quellsystemen und werden nur zur Zeit der Anfrage aus diesen geholt. Die Daten werden dabei während der Anfragebearbeitung vom lokalen Schema der Datenquelle in das globale Schema des Integrationssystems transformiert. Dies entspricht einem *Pull-Prinzip*. [24]

  Nach der Anfrage werden die Daten meist wieder verworfen, es können aber auch Caching Verfahren angewandt werden. Somit existiert der Datenbestand nur *virtuell*. [24]

  Im Gegensatz zur materialisierten Integration, sind Anfragen in virtuellen Integrationssystemen meist teuer. Dies kommt daher, dass die globalen Anfragen zunächst in einzelne Anfragepläne an die jeweiligen Quellsysteme zerlegt werden, die Daten aus den

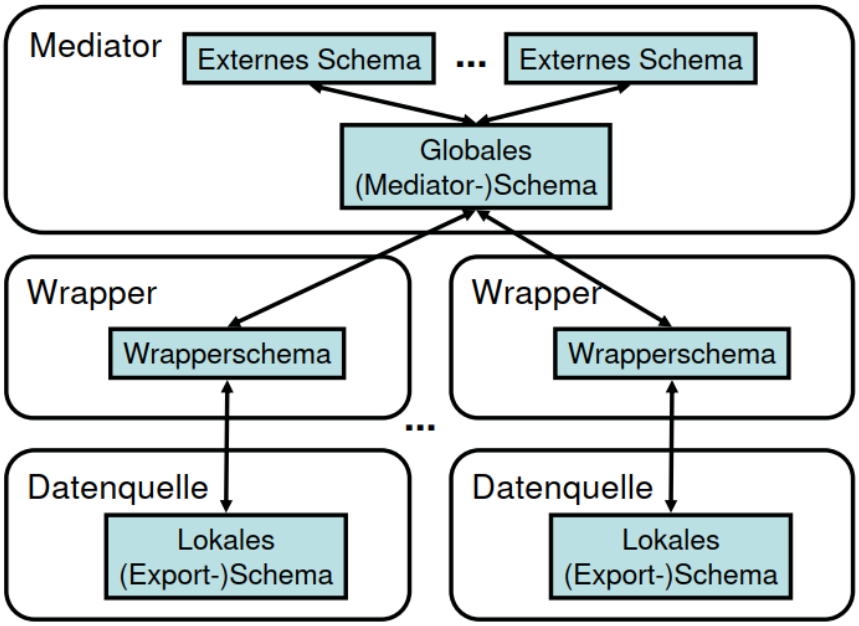

Abbildung 8: Diagramm einer Mediator-Wrapper Architektur [24, S.97]

Systemen geholt und transformiert werden und ggf. vereint werden müssen. Auch die Problematik von temporär nicht erreichbaren Quellsystemen existiert. [24]

Im Laufe der Zeit sind mehrere architekturelle Ansätze für Integrationssysteme entstanden. Für den Zweck dieser Arbeit werden vor allem Mediatorbasierte Systeme und die Peer-Data-Management-Systeme von besonderem Interesse tiefer betrachtet.

- **Mediator-Wrapper Architektur:**

  Als Verallgemeinerung vorheriger Architekturen wurde erstmals von Wiederhold eine Mediator-Wrapper Architektur vorgeschlagen. Diese identifiziert zwei Rollen: Wrapper für den Datenzugriff und Mediatoren, welche auf die Wrapper zugreifen und eine strukturelle und semantische Integration der Daten vornehmen. [24]

  Dabei werden die Daten aus dem lokalen Schema ihrer Datenquellen in das vom Mediator verwaltete globale Schema überführt. Jeglicher Zugriff auf die verwalteten Daten erfolgt dabei über den Mediator, wodurch eine vollständige Transparenz für den Nutzer erreicht wird, wie **Abb. 8** demonstriert. [24]

  Dabei definiert Wiederhold Mediatoren wie folgt[1]:

  Ëin Mediator ist eine Softwarekomponente, die Wissen über bestimmte Daten benutzt, um Informationen für höherwertige Anwendungen zu erzeugen."

  Um einen entsprechenden Mehrwert aus den Daten ziehen zu können, muss bei der Implementierung des Mediators auf Expertenwissen aus der jeweiligen Domäne zurückgegriffen werden. Wiederhold schlägt dabei vor, dass Mediatoren klein und einfach gehalten werden sollten, um die Wartbarkeit zu erhöhen. [24]

[1] Übersetzung entnommen aus [24, S.98]

Die eigentliche Kommunikation mit den Datenquellen wird von den Wrappern übernommen. Diese sind Softwarekomponenten, welche jeweils für den Datenaustausch mit einer bestimmten Quelle spezialisiert sind. Ihre Aufgabe ist es, die Daten aus dem lokalen Schema des Quellsystems in das globale Schema des Integrationssystems. [24]

Dabei überwinden die Wrapper in ihrer Funktion technische-, Schnittstellen-, Datenmodell- und schematische Heterogenität. Wie auch bei den Mediatoren, ist es wünschenswert die Wrapper so einfach wie möglich zu halten. [24]

Folgendes sind dabei die Qualitätskriterien, die für einen Wrapper in Frage kommen:

- **Schnelle Implementierung:**
  Es soll möglich sein, einen einfachen Wrapper innerhalb weniger Stunden zu implementieren. [24]
- **Erweiterbarkeit:**
  Aufgrund möglicher Änderungen an der Datenquelle selbst, sollte es möglich den Wrapper schnell auf diese Änderungen anzupassen.
- **Wiederverwendbarkeit:** [24]
  Eine einfache Wiederverwendung von Wrappern sollte möglich, um damit im Laufe der Zeit eine Wrapperbibliothek aufbauen zu können. [24]
- **Wartbarkeit:**
  Aufgrund der verteilten Natur der Datenquellen in Integrationsszenarien, sollte es möglich sein, dass Datenquellen ihre eigenen Wrapper entwickeln und warten können. [24]

- **Peer-Daten-Management-Systeme**

  Bei Peer-Daten-Management-Systemen(PDMS) wird die Trennung zwischen Datenquelle und Integrationssystem aufgehoben. In Anlehnung an Peer-to-Peer Netzwerke, wird jedes angebundene System als gleichwertiger Peer im Gesamtsystem betrachtet. Ein Peer erfüllt gleichzeitig die Rolle einer Datenquelle und eines Mediators. [24]

  Dabei beantwortet er Anfragen mit Daten aus seinen lokalen Datenbeständen und reicht in seiner Rolle als Mediator Anfragen weiter an andere Peers. Es entsteht ein Netzwerk aus Datenquellen. Um dies zu erreichen, müssen die Peers jeweils Schema Mappings bei sich speichern, welche äquivalente Elemente zwischen dem lokalen Schema und den Schemata der anderen Peers. Auch wie die Daten bei der Übetragung von einem Peer zum anderen transformiert werden müssen, ist in dem Mapping enthalten. [24]

  Neber Orts- und Schematransparenz, verfügen die PDMS über eine hohe Flexibilität was das Hinzufügen und Entfernen von Datenquellen angeht. Für jede neue Datenquelle muss nämlich lediglich nur ein Schema Mapping definiert werden. [24]

Durch das Wegfallen eines globalen Schemas, wird das System auch nicht durch das Hinzufügen und Entfernen von Datenquellen beeinträchtigt. Es müssen nur Mappings erzeugt bzw. entfernt werden. Dies führt dazu, dass dem System vorher nicht alle Datenquellen bekannt sein müssen, da sich unbekannte Datenquellen jederzeit über ein neues Schema Mapping anbinden lassen. [24]

Demnach wissen Anwendungen auch nicht aus welchen Systemen die Daten für eine Anfrage gekommen sind, ausser es wurden ausführliche Metadaten über die Quellsysteme, aus denen die Daten ursprünglich stammen, mitgeschickt. [24]

Durch die verteilte Natur von PDMS kommt es zu einer komplexen Anfragebearbeitung, welche sich über mehrere Peers hinweg strecken kann. Die Anfragen müssen von einem Peer zum nächsten umformuliert werden und es muss darauf geachtet werden, dass es zu keinen Zyklen kommt. Ebenso kann die vielfache Transformation der Daten dazu führen, dass es zu einem Verlust ans semantischer Information und Qualität kommt. [24]

2.2.6 Semantische Integration

Bei der semantischen Integration wird versucht die semantische Heterogenität der Datenquellen zu überbrücken. Die verschiedenen Abstufungen einer semantischen Integration und ein konkrete Vorgehensweise zur semantischen Integration unter der Verwendung von Ontologien sollen in diesem Abschnitt weiter vorgestellt werden.

Datenfusion

Eine der Aufgaben der semantischen Integration ist die Datenfusion. Das Ziel der Datenfusion ist die Vereinigung von Daten aus verschiedenen Quellsystemen in einer integrierten Antwort. [24]

Ein wichtiger Aspekt davon ist die Erkennung von Duplikaten, also der verschiedenen Repräsentationen von gleichen Realweltobjekten, und die Integration dieser in eine einheitliche, konsistente Repräsentation. [24]

Die Art dieser Integration lässt sich in drei verschiedene Abstufungen unterscheiden:

- **Vereinigung:**
  Die einfachste Stufe, bei der lediglich Daten unterschiedlicher Quellen zusammengefügt werden. Es findet keine semantische Integration im eigentlichen Sinne statt. [24]
- **Anreicherung:**
  Auf dieser Stufe findet keine vollständige Integration der Daten statt, aber eine Anreicherung der Daten durch Metadaten. So können beispielsweise potenzielle Duplikate und Widersprüche gekennzeichnet werden. Die weitere Integration unter Verwendung dieser Metadaten ist dann dem Benutzer überlassen. [24]
- **Fusion:**
  Dies stellt die höchste und zugleich aufwendigste Stufe der Integration dar. Es findet

eine automatische Bereinigung und vollständige Integration semantisch gleicher Dinge statt. [24]

Ontologiebasierte Integration

Bei dieser Variante der Integration wird versucht unter der Verwendung von *Ontologien* die semantische Heterogenität zu überbrücken. Diese Ontologien sind formale Modelle eines spezifischen Anwendungsbereichs, welche so genau spezifiziert wurden, dass man mittels logischer Interferenz über dieses Modell die semantische Heterogenität überbrücken kann. [24]

Ontologien werden mittels *Beschreibungslogiken (description logics)* aufgestellt und besitzen für eine Integration nützliche Eigenschaften:

- Die Definition eines Vokabulars für eine bestimmte Domäne
- Die Strukturierung von Beziehungen zwischen den Begriffen des Vokabulars
- Beihilfe zur Verständigung innerhalb einer größeren Menge von Personen

In ihrem Grundwesen beschreiben Ontologien *Konzepte*. Diese Konzepte stehen ein für bestimmte Dinge oder eine Klasse von Dingen. Dabei genügt es nicht eine einfache Wortliste anzugeben, vielmehr wird ein detailliertes Modell benötigt, welches Eigenschaften der beschriebenen Konzepte, wie Attribute, Integritätsbedingungen und Beziehungen zwischen Konzepten. [24]

Ontologien eignen sich demnach besonders gut für den Austausch von Daten, besonders wenn eine größere Gruppe von Personen oder Organisationen am Austausch beteiligt ist. So können beispielsweise Ontologien für Unternehmen aufgebaut werden, die alle für den Unternehmenszweck wichtigen Begriffe erfasst und definiert. Aber auch Standards lassen sich gut mittels Ontologien ausdrücken, wenn diese eine bestimmte Domäne abdecken sollen. [24]

Eine ontologiebasierte Informationsintegration erfolgt dabei in drei Schritten: [24]

1. Als Erstes muss eine globale Ontologie mit allen wesentlichen Konzepten der Domäne erstellt werden. Diese funktioniert dann auch gleichzeitig als das globale Schema der Integration.
2. Anschliessend werden die Datenquellen anhand der globalen Ontologie modelliert. Dabei werden ihre Relationen und Attribute als abgeleitete Konzepte aus dieser beschrieben. An dieser Stelle kann es auch notwendig sein, die globale Ontologie zu erweitern. Dies ist bspw. der Fall, wenn eine Datenquelle nachträglich an das System angeschlossen werden soll.
3. Sind diese beiden Schritte erfolgt, dann ist es möglich Anfragen an das Integrationssystem zu senden. Diese werden dabei ebenfalls als Konzepte der globalen Ontologie formuliert.

Damit lassen sich Domänen weitaus genauer spezifizieren, als dies mit relationalen Datenmodellen möglich wäre. [24]

Ontologien lassen sich grundsätzlich in zwei verschiedenen Arten einteilen:

- **Top-Level-Ontologien:**

  Bei dieser Variante handelt es sich um Ontologien, welche fundamentale Begriffe definieren und zueinander in Bezug setzen. [II.276] Ihr Zweck ist es ein Grundgerüst zu definieren, in welchem allgemeine Sachverhalte, wie beispielsweise Klassen, definiert sind. Auf diesen kann dann für den Aufbau von anderen Ontologien aufgesetzt werden. [24]

- **Domänenspezifische Ontologien:**

  Diese sind Ontologien, welche sich auf einen spezifische Anwendungsbereich spezialisiert haben, und diesen versuchen zu modellieren. [24]

Aus dem Grundgedanken der semantischen Integration, ist das *Semantic Web* entstanden. Dessen Ziel ist es, das Word Wide Web als ein Informationssystem zu betrachten. Dabei ist dies mehr als eine Vision und keine konkrete Technik zu betrachten. Im Zuge dieser Vision wurden durch das W3C in den letzten Jahren mehrere verschiedene Techniken standardisiert. Eine davon ist das Resource Description Framework (RDF). Dieses ist ein Datenmodell zur Beschreibung von Objekten und deren Eigenschaften. [3]
RDF besteht zunächst aus drei Grundelementen: [10]

- **Ressourcen:**
  "Dinge"über welche man im Datenmodell sprechen will. Ressourcen werden über eine URI adressiert.

- **Eigenschaften:**
  Beschreiben mögliche Eigenschaften von Ressourcen und sind ebenfalls Ressourcen.

- **Aussagen:**
  Beschreiben Eigenschaften von Ressourcen anhand eines Subjekt-Prädikat-Objekt Prinzips. Aussagen bestehen aus einem Tripel (R,A,V), welches angibt, dass Ressource R eine Eigenschaft A mit dem Wert V besitzt. Werte können dabei selbst andere Ressourcen oder nicht weiter interpretierbare Zeichenketten sein.

Auch wenn RDF mit dem Hintergrund des Semantic Web entwickelt worden ist, ist es zunächst ein vom Web unabhängiges Datenmodell, aber beschreibt vor allem im Web erreichbare Ressourcen. [24]

An sich werden mit RDF nur Daten auf Instanzebene beschrieben. Dabei gibt es keine Einschränkungen bei den verwendeten Bezeichnern oder der Verwendung von Typen. Damit sind RDF-Daten vorerst schemalos, was für die Integration ein großes Hindernis darstellt, da eine computerbasierte Analyse von einer potenziell unbeschränkten Semantik nicht möglich ist. Um dieses Problem zu lösen, muss zunächst die Struktur und die erlaubten Bezeichner eingeschränkt werden, also ein Schema erstellt werden, welches verwendet werden soll. [24]

Für diesen Zweck wurde die *RDF Vocabulary Description Language* (historisch bedingt abgekürzt mit RDFS) erstellt. Damit erlaubt sich u.a. die Spezifikation von Klassen für Ressourcen, welche dann als Typbezeichner verwendet werden können (*rdfs:Class*). Ebenefalls kann festgelegt werden, welche Eigenschaften diese Klassen besitzen dürfen (*rdf:Property*). So wie die erlaubten Klassen für Subjekte und Objekte in in Aussagen (rdfs:range). [5]

Mit diesen Hilfsmitteln ist es möglich eine objekt-orientierte Modellierung von Ontologien auszuführen. Im Gegensatz zur traditionellen Objekt-Orientierung, haben in RDFS erstellte Ontologien, die Eigenart, dass die Definition einer Eigenschaft losgelöst ist von der Definition einer Klasse. Diese implizite Zugehörigkeit von Eigenschaften zu ihren Klassen führt dazu, dass sich RDFS-Modelle beliebig erweitern lassen oder durch Integration von anderen RDFS-Modellen anreichern lassen. [10]

RDF selbst gibt dabei nur eine Spezifikation vor. Eine Implementierung dieses Standards in XML wurde mit RDF/XML umgesetzt. [4] Eine weitere Implementierung findet sich in JSON-LD, welches RDF mittels der Syntax von JSON umsetzt. [1]

2.3 Verwandte Arbeiten

In diesem Kapitel sollen einige verwandte Arbeiten und aktuelle Forschungsergebnisse aus dem Bereich des PLM vorgestellt werden. Die hier vorgestellten Arbeiten stellen dabei Erweiterungen zum traditionellen PLM Ansatz dar und stehen deswegen in direktem Bezug zu dieser Arbeit.

2.3.1 Semantic Data Management

Abramovici et al. prägten in einer Reihe von Arbeiten den Begriff des *Semantic Data Managements* (SDM). SDM gilt als Erweiterung des traditionellen PLM Ansatzes, da die Forscher erkannten, dass dieser nicht mehr den Anforderungen der Entwicklung von *Smart Products* (SP) entspricht. [7]

Bei SP handelt es sich um intelligente cyber-physische Systeme, welche über das Internet mit Services oder anderen cyber-physischen Produkten interagieren. Diese Wechselwirkungen werden nicht vom traditionellen PLM Ansatz abgedeckt. Ebenso werden Wechselwirkungen eines Produktes zu seinen *Digitalen Zwillingen*, also Rückflüsse von Informationen, wie beispielsweise Sensordaten, einer realen Instanziierung des Produktes in PLM nicht berücksichtigt. [7]

Um Lösungen für diese Problematiken zu finden, identifizierten die Forscher zunächst drei verschiedene Arten von Informationen, welche bei der Entwicklung von SP auftauchen: [7]

- **Architekturelle Informationen**, welche die Zusammensetzung der verschiedenen Komponenten eines SP beschreiben, sowie deren Wechselwirkungen.
- **Komponenten Informationen**, welche die einzelnen Komponenten, sowie ihre Funktionsweisen beschreiben.
- **SP Benutzungs Informationen**, also Daten die während des realen Betriebs des SP anfallen, wie beispielsweise Sensordaten.

Das Sammeln und die Integration dieser Informationen in einem cross-enterprise Netzwerk, wurden als die größten Probleme in der Verwaltung von für SP relevanten Informationen identifiziert. [7]

Der von den Forschern vorgeschlagene SDM Ansatz erweitert PLM um die Verwaltung dieser drei Informationstypen und berücksichtigt dabei die Entwicklung, Verwaltung von IT Services und die Wechselwirkung zwischen IT Services und mechatronischen Komponenten.

Da dies in einem cross-enterprise Netzwerk geschieht und eine effiziente Kollaboration verschiedener Unternehmen stattfinden soll, wird ein durchgängiger und transparenter Datenfluss benötigt. Da Unternehmen üblicherweise ihre eigenen Datenmodelle aufbauen, muss diese Heterogenität zunächst überbrückt werden. [7]

Die Forscher setzten dabei auf den Einsatz einer ontologiebasierten Integration. Dafür wurde im Rahmen von SDM die *SDM Ontologie* entwickelt. Diese ermöglicht es die Konzepte von Organisationen, SP, Instanzen von SP, sowie Resourcen auszudrücken. [7]

Für die Implementierung des SDM wurde eine Integrationsplattform aufgebaut. Die Anbindung verschiedener Datenquellen wurde anhand einer Micro-Service Architektur realisiert. Diese hatte den Vorteil, dass die einzelnen Services unabhängig voneinander agierten und somit die Entwicklung dieser Services unabhängig voneinander ablaufen kann, ohne dass das restliche System davon beeinträchtigt wird. [7]

Für die Speicherung der Daten wurde die Graphdatenbank Neo4j als Backbone eingesetzt. Diese wurde verwendet, da das Datenmodell von SDM selbst bereits einen Graphen bildet, und sich damit für Techniken des *Graph Traversal* eignet, welche von Graphdatenbanken zur Verfügung gestellt werden. [8]

In einer weiteren Arbeit konnte dieser Ansatz erweitert werden, indem verschiedene semantische Techniken auf diese Integrationsplattform angewendet wurden. So wurde ein Text Mining Ansatz verwendet, um an das System angebundene Dokumente auf verschiedene Schlüsselwörter zu durchsuchen. Die Resultate der Text Analyse wurden als eigene Nodes im Graphen an die betreffenden Nodes der Dokumente angefügt. Diese Resultate wurden anschließend für die Auswertung von Anfragen verwendet. [8]

Dadurch wurde es ermöglicht effiziente Anfragen in der Form von natürlicher Sprache an das System zu stellen. Diese Eigenschaft war für Nutzer des Systems von großer Wichtigkeit, da diese somit keine besonderen Technologien, wie beispielsweise SPARQL, für das Stellen von Anfragen beherrschen müssen. [8]

2.3.2 Ontologien für PLM

Der Einsatz von Ontologien eignet sich besonders gut für die Erschaffung von Standards. [24]

Ein Versuch solch einen Standard für PLM zu erschaffen ist das *Open Services for Lifecycle Collaboration* (OSLC). OLSC ist dabei eine offene Community mit namhaften Vertretern aus der Industrie, welche versucht eine offene Spezifikation für PLM und andere damit verbundenen Gebiete mit Hilfe von RDF abzubilden. [34]

OSLC gibt dabei nur eine Spezifikation vor. Diese müsste zunächst von Tools implementiert werden. Die Spezifikation für PLM befindet sich dabei noch im Status *Draft* und ist demnach vorerst nicht für Zwecke innerhalb dieser Arbeit einsetzbar. [34]

2.3.3 Stand der Industrie

Da die meisten Unternehmen nicht oft Informationen über ihre konkreten PLM Umsetzungen öffentlich machen, ist es sehr schwierig einen Überblick über den aktuellen Stand der Integration der IT-Applikationen zu erhalten.

Eine Suche im Internet ergab eine im Jahr 2016 durchgeführte Studie der IDG Business Media GmbH über den Einsatz von Business Analytics in deutschen Unternehmen. Da eine gelungene Integration von Applikationen eine Voraussetzung für diese Analyse Verfahren ist,

wurden IT-Experten verschiedener namhafter deutscher Unternehmen nach ihrem gefühlten Stand der Datenqualität und Integration in ihren Unternehmen gefragt. [23]

Es kam dabei heraus, dass die Entwicklungs- und Produktionsabteilungen dabei jeweils am wenigsten Daten innerhalb der Unternehmen sammeln. Die befragten IT-Experten sehen ihre Firmen dabei nicht gut vorbereitet für die Datenintegration. Ernüchternd ist das Resultat, dass 89% der Befragten Teilnehmern angaben, in ihren Unternehmen den Datenaustausch per E-Mail vorzunehmen. [21]

Durch einen Experten im PLM Umfeld wurde auf ein gemeinsames Projekt der Firma Schaeffler und IBM aufmerksam gemacht. Diese bauen mit dem *Engineering Cockpit* eine Integrationsplattform für Daten und Prozessverwaltung innerhalb von Unternehmen. Dabei soll das Engineering Cockpit sich als zusätzliche Schicht über die Autoren und PLM-Systeme stellen. Die Integration der Datenquellen beschränkt sich dabei auf Systeme zur Projektverwaltung sowie PLM Systeme und andere im Bereich der mechanischen Konstruktion verwendete Applikationen. [23]

Leider konnten keine weiteren Informationen über die konkrete Realisierung der Anbindungen erhalten werden.

2.3.4 Einordnung dieser Arbeit

Diese Arbeit befasst sich mit dem Problem, der Integration verschiedener Datenquellen und Applikationen im PLM Umfeld. Da die in diesem Kapitel beschriebenen Arbeiten einen direkten Bezug zur Integration von verschiedenen Datenquellen im PLM Gebiet haben, soll hier eine kurze Eingrenzung und Bewertung stattfinden, in welcher Weise diese Arbeit sich zu den hier vorgestellten Arbeiten positioniert.

MBSE stellt ein hochaktuelles und relevantes Forschungsgebiet im PLM dar. Für eine erfolgreiche Verwendung von MBSE ist eine Integration von einer Anzahl an zusätzlichen Applikationen erforderlich, welche nicht im traditionellen PLM verwaltet werden. In Zuge dessen, nimmt diese Arbeit eine unterstützende Funktion für MBSE ein, und versucht eine leichtgewichtige Möglichkeit zur Integration von verschiedensten Applikationen zu finden. Ebenfalls wurden im Rahmen von MBSE die fehlenden Verknüpfungen zwischen Strukturmodellen angesprochen. Diese treten durch Medienbrüche zwischen den einzelnen Applikationen auf. Eine Überbrückung dieser Medienbrüche ist ein weiteres Problem, dem diese Arbeit sich widmen will. Dabei ist zu beachten, dass diese Arbeit dabei nicht auf die von MBSE erstellten Systemmodelle zurückgreift, sondern zunächst versucht auf technischer Ebene Lösungen für diese Probleme zu finden.

SDM ist eine fortgeschrittene Erweiterung von PLM für das Systems Engineering, und greift, unter der Verwendung der eigens erstellten Semantik, auf eine Vielzahl von semantischen Techniken zurück. Dabei wurde ebenfalls eine Integration verschiedener Datenquellen vorgenommen. Der Fokus der Integration lag dabei auf den Entwicklungsdaten und realen Daten der Smart Products bzw. Nutzungsdaten selbst.

Diese Arbeit strebt es an, eine generische Architektur für Datenintegration zu bilden, bei dem sowohl Quellen aus dem Smart Engineering angebunden werden können, als auch Datenquellen aus allen anderen möglichen Bereichen. Stattdessen ließe sich beispielsweise die Ontologie von SDM verwenden, und die in dieser Arbeit konzipierte Architektur als Integrationslayer einsetzen. Ebenso ist eine Verwendung von OSLC vorstellbar, sobald dies einen fertigen Status erreicht.

3 Design des Frameworks

In diesem Kapitel soll das Design des für das *Semantic Product Information and Digitized Engineering Repository* (kurz: *SPIDER*) erarbeitet werden. Dabei wird zunächst ein Schlüsselszenario betrachtet, welche als Grundlage für die Ableitung von Use Cases sowie Anforderungen dienen soll.

Aufbauend darauf werden die konkreten Anforderungen für die Integration aufgestellt sowie Überlegungen zur Anbindung der Datenquellen getätigt.

Zusätzlich findet eine Betrachtung einiger im PLM Umfeld eingesetzter Systeme statt. Aus den daraus entnommenen Erkenntnissen wird eine Ontologie für die semantische Beschreibung von Datenquellen angefertigt.

Anschließend wird das Design der Architektur von SPIDER und seiner Komponenten besprochen.

3.1 Schlüsselszenarien

In diesem Abschnitt sollen einige Schlüsselszenarien näher beschrieben werden, welche als die Grundlage für die Entwicklung des Systems in dieser Arbeit dienen sollen.

3.1.1 Einführung der Integrationsplattform

Dieses Szenario beschreibt die Herausforderungen und Anforderungen, die bei der Einführung einer Integrationsplattform, welche als administratives Backbone für die Systemlandschaft eines Unternehmens eingesetzt werden soll.

Viele Unternehmen haben sich im Laufe der Zeit bereits grössere Systemlandschaften, bestehend aus verschiedenen IT-Applikationen, aufgebaut. Die Aufgabe dieser Systeme ist es, die Prozesse innerhalb der verschiedenen Bereiche, wie bspw. Entwicklung oder Controlling, zu unterstützen. Diese Systeme unterscheiden sich sowohl stark in ihren Arbeitsweisen, als auch in ihren Aufgabengebieten. Demnach sind auch die verschiedenen Disziplinen auf unterschiedliche, auf Ihre Zwecke zugeschnittene IT-Tools angewiesen. So ist die Konstruktion auf den Einsatz von PDM-Systemen für die Verwaltung ihrer Produktstrukturen angewiesen. Die Softwareentwicklung hingegen benötigt Software-Versionierungssysteme.

Alle diese Anwendungen unterscheiden sich sowohl in ihren verwalteten Daten, als auch in den Möglichkeiten diese Daten abzufragen. Eine erfolgreiche Integrationslösung sollte demnach diese Unterschiede alle möglichst unter einem einheitlichen Datenformat sowie Interface verstecken. Ebenfalls soll es sowohl für interne als auch unternehmensfremde Anwender möglich sein erfahren zu können, welche Datenquellen am System angeschlossen sind, welche Methoden diese zur Verfügung stellen und welche Daten diese zur Verfügung stellen.

Anwender können dabei menschliche Benutzer des Systems sein, die mittels eines Clients Informationen aus den Datenquellen anfordern oder andere Applikationen, die automatisiert mit dem System kommunizieren wollen, um bspw. automatisierte Prozesse durchzuführen.

Dabei soll es den Anwernden möglich sein typische CRUD-Zugriffe auf die Daten einer Datenquelle auszuführen. Diese sind Create/Erstellen, Read/Lesen, Update/Verändern, Delete/-Löschen von Daten.

Von den Datenquellen erhaltene Daten sollen dabei in einem einheitlichen Format übertragen werden. Nach Möglichkeit, sollen Anwender Zusatzinformationen über die in diesem Format definierten Elemente erhalten können.

Die Anbindung der Datenquellen selbst wird von den Betreibern des Systems vorgenommen. Ihre Aufgabe ist es die Datenmodelle der Datenquellen zu beschreiben sowie diese Datenquellen mit ihren Datenmodellen an dem System zu registrieren.

Die Anbindung einer neuen Datenquelle soll dabei keine negativen Auswirkungen auf den restlichen Betrieb des Systems haben. Dabei sollen für die Betreiber Möglichkeiten zur Verfügung gestellt werden, um eine Anbindung intuitiv durchführen zu können.

3.1.2 Reagieren auf Änderungen

Dieses Szenario baut auf dem vorherigen auf und beschreibt die Auswirkungen die Änderungen innerhalb der Datenquellen auf die Integrationsplattform haben sollen.

Durch den stetigen Wandel der Anforderungen an ein Unternehmen, treten oft Änderungen an den eingesetzten Anwendungen ein. Dabei können sich die Änderungen nur innerhalb einer Datenquelle selbst abspielen, wie bspw. Änderungen am Datenmodell. Oder aber der Austausch oder das Entfernen von ganzen Datenquellen an sich.

In diesen Fällen müssen die Betreiber des Systems in der Lage sein, diese Änderungen in SPIDER einspielen zu können. Änderungen sollen dabei keine negativen Auswirkungen auf das restliche System besitzen.

3.2 Anforderungen

In diesem Abschnitt sollen die Anforderungen anhand der vorgestellten Schlüsselszenarien abgeleitet.

3.2.1 Stakeholder

Aus der Betrachtung der Problemstellung und der Schlüsselszenarien lassen sich zunächst folgende Stakeholder des Systems identifizieren:

Betreiber des Systems:

Dies sind unternehmensinterne Domänenexperten, welche für die Konfiguration des Backbones und der unterliegenden Datenquellen zuständig sind. Diese sollen die lokalen Datenmodelle der Datenquellen festlegen und die an der Integration teilnehmenden Datenquellen am Backbone registrieren.

Benutzer:

Dies können unternehmensinterne oder externe Programmierer oder Applikationen sein, welche die Funktionalität des Systems in Anspruch nehmen wollen und beispielsweise Daten aus einer Datenquelle anfordern wollen. Benutzer sollen dabei, außer den grundlegenden Kenntnissen um auf das System zugreifen zu können, keine weiteren Kenntnisse über die unterliegenden Datenquellen oder das Datenmodell selbst besitzen müssen um mit dem System zu interagieren.

Die einzelnen Use Cases der Stakeholder sind in **Abb. 9** abgebildet und werden im Folgenden weiter erläutert.

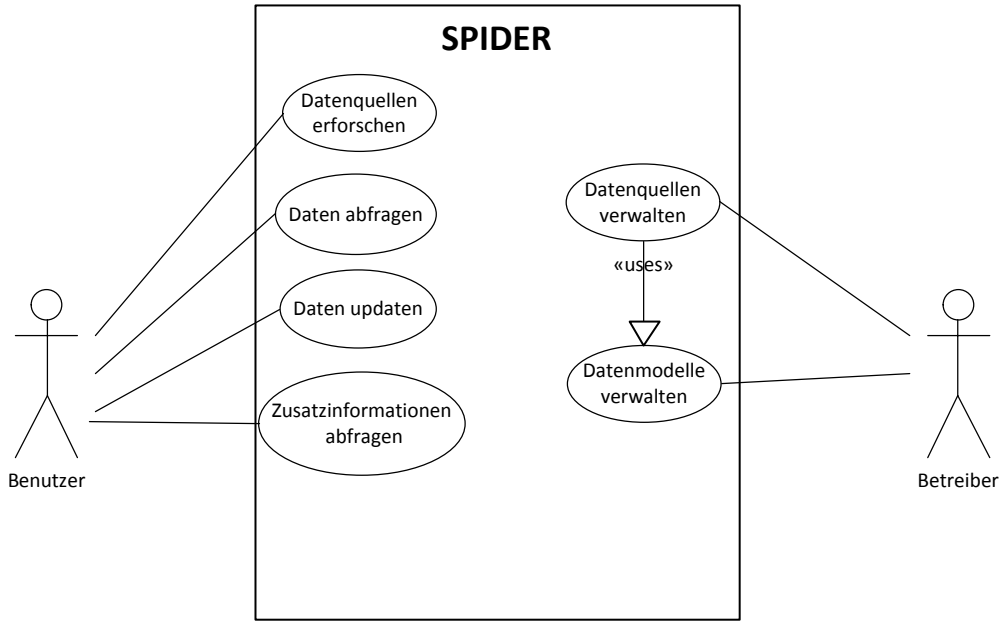

Abbildung 9: Use Case Diagram für die einzelnen Stakeholder

3.2.2 Use Cases

In diesem Abschnitt findet eine Auflistung von den erstellten Use Cases und ihrer Beschreibung in tabellarischer Form statt.

Use Case	1.1
Name	Datenmodelle verwalten
Aktoren	Betreiber
Beschreibung	Dem Betreiber soll es möglich sein Datenmodelle im System einzutragen und diese, bei Bedarf, anzupassen. Dies umfasst Klassendefinitionen sowie die dazugehörigen Attribute. Die Datenmodelle dienen der Beschreibung von Datenquellen sowie als Schemata für den Austausch von Daten mit dem System.

Use Case	1.2
Name	Datenquellen verwalten
Aktoren	Betreiber
Beschreibung	Dem Betreiber soll es möglich sein, neue Datenquellen an das System anschließen zu können, sowie vorhandene Datenquellen anzupassen oder zu entfernen. Jeder Datenquelle soll ein Datenmodell zugewiesen werden, welches bei Bedarf ebenfalls angepasst werden kann. Es ist dabei unerheblich um welche Datenquelle es sich handelt, das System soll also in der Lage sein jede mögliche Applikation zu integrieren.

Use Case	2.1
Name	Datenquellen erforschen
Aktoren	Benutzer
Beschreibung	Beschreibung: Der Benutzer kann Anfragen an das System stellen, um Informationen über die angebundenen Datenquellen zu erhalten. Zu diesen Daten gehören: • Name der Datenquelle • Endpunkt der Datenquelle • Informationen über das Datenmodell (dies beinhaltet Informationen über die Klassen und deren Attribute, welche von der Datenquelle verwaltet werden)

Use Case	2.2
Name	Daten abfragen
Aktoren	Benutzer
Beschreibung	Ist eine Datenquelle bekannt, so soll ein Benutzer Daten aus dieser Datenquelle abfragen können. Ausser dem verwendeten Datenmodell der Datenquelle muss der Benutzer dabei über keine weiteres sonstiges Wissen über die Datenquelle oder ihre Interna benötigen. Zugriffe auf die Daten der Datenquelle sollen alle auf eine vereinheitlichte Art und Weise erfolgen. Ist die Anfrage erfolgreich, erhält der Benutzer die Daten ebenfalls in einer vereinheitlichten Form zurück.

Use Case	2.3
Name	Daten updaten
Aktoren	Benutzer
Beschreibung	Ist eine Datenquelle bekannt, sowie eines ihrer verwalteten Objekte, dann soll es dem Benutzer möglich sein, Änderungen an diesem Objekt durchzuführen. Dies soll über ein vereinheitlichtes Interface geschehen, welches für alle Datenquellen gleich ist. Dabei wird vom Benutzer angegeben welche Art der Änderung durchzuführen ist (Editieren, Hinzufügen, Löschen). Sowie die Attribute und die Werte, die sie erhalten sollen. Die Datenquelle überprüft daraufhin ob die angeforderte Änderung erlaubt ist. Wurde die Änderung nicht durchgeführt, wir dies dem Benutzer durch eine Fehlermeldung signalisiert.

Use Case	3.1
Name	Zusatzinformationen abfragen
Aktoren	Benutzer
Beschreibung	Ein Benutzer soll die Möglichkeit besitzen Informationen zu Objekten abzufragen, die ihre Definition betreffen. Das umfasst die darauf erlaubten Operationen, die Definitionen der Attribute, sowie Datentypen und eventuelle Verwendungszwecke er definierten Attribute und Klassen.

3.2.3 Qualitätskriterien

In diesem Abschnitt sollen die Qualitätskriterien festgelegt werden, nach denen eine erfolgreiche Lösung bewertet werden soll.

- **Korrektheit:**

  Korrektheit im Sinne von SPIDER bedeutet, dass auf alle Anfragen an die Datenquellen, die richtigen Daten zurückgeliefert bzw. die richtigen Operationen in der Datenquelle ausgeführt werden.

- **Anbindungen der Quellsysteme:**

  Neue Quellsysteme sollen an SPIDER angeschlossen werden können, ohne dass dabei Änderungen am SPIDER selbst vorgenommen werden müssen. Ebenfalls soll es keine negativen Auswirkungen auf den Rest des Systems haben, wenn neue Datenquellen hinzugefügt werden. Eine Anbindung soll dabei möglichst schnell und einfach durchzuführen sein. Es soll dabei unerheblich sein um, was für Systeme es sich handelt.

- **Transparenz:**

  Angebundene Quellsysteme sollen durch SPIDER abstrahiert werden. Benutzer des Systems sollen demnach mit den Quellsystemen interagieren können, ohne Wissen über deren Datenmodelle oder interne Abläufe zu benötigen. Demnach sollen Änderungen an den Datenquellen für die Benutzer komplett unsichtbar sein.

- **Wartbarkeit:**

  Es ist eine wichtige Anforderung von SPIDER auf Änderungen innerhalb der Datenquellen reagieren zu können. Änderungen sollen sich dabei nicht negativ auf das restliche System auswirken. Dies umfasst Änderungen innerhalb einer Datenquelle, als auch das Hinzufügen, Austauschen oder Entfernen von Datenquellen.

- **Wiederverwendbarkeit:**

  SPIDER ist nicht für den Einsatz innerhalb eines konkreten Unternehmens ausgelegt, sondern als generische Integrationsplattform angedacht, die in mehreren Unternehmen eingesetzt werden kann, unabhängig von deren bereits eingesetzten IT-Landschaften.

  Aber auch im Hinblick auf die internen Komponenten von SPIDER soll ein hoher Grad an Reuse angestrebt werden, wie bspw. die Wiederverwendung von Komponenten, welche die Anbindung zu Datenquellen betreffen.

3.3 Integration der Datenquellen

Da es sich um ein Datenintegrationsszenario handelt, muss geklärt werden, wie die Integration der Datenquellen stattfinden soll. Deswegen sollen in diesem Unterkapitel benötigte Entscheidungen über die gewünschte Autonomie und Transparenz der Datenquellen getroffen werden. Außerdem sollen Lösungsansätze besprochen werden, um die Heterogenitäten der einzelnen Datenquellen zu bewältigen.

3.3.1 Ausmaß der Integration

In diesem Abschnitt soll festgelegt werden, welche Form der Integration der Datenquellen in SPIDER angewendet werden soll. Dafür werden Fragen der Autonomie der Quellsysteme, gewünschte Transparenz sowie der Datenfusion und Materialisierung der Daten untersucht und beantwortet.

Einschränkungen der Autonomie:

Eine Einschränkung der Autonomie bei der Entwicklung, also Design- und Schnittstellenautonomie, der Datenquellen ist größtenteils nicht möglich, da diese entweder bereits vorhanden sind oder proprietär entwickelte Systeme sind.

Lediglich bei der Entwicklung von unternehmensinternen Systemen wären Einschränkungen bei der Wahl der Interfaces und Datenmodelle möglich, aber dies kann aufgrund von unternehmensinternen Gegebenheiten nicht gewährleistet werden. Dies betrifft ebenfalls die Weiterentwicklung der Datenquellen.

Ebenfalls sollen die Quellsysteme selbst weiterhin ihren gewünschtem Zweck nachgehen können, ohne dass die Benutzer dieser Systeme Einschränkungen in ihren Betriebsabläufen in Kauf nehmen müssen.

Aus diesen Gründen muss also ein hohes Ausmaß an Autonomie der Datenquellen für die Entwicklung von SPIDER berücksichtigt werden.

Erforderliche Transparenz:

Hier soll eine Bewertung über die verschiedenen benötigten Ausprägungen der Transparenzkriterien erfolgen, und wie sich diese auf die Entwicklung des Systems auswirken.

- **Ortstransparenz:**

  Die Frage die sich bei dieser Art der Transparenz stellt, ist ob der physikalische Ort einer Datenquelle im Netzwerk vor dem Benutzer versteckt werden soll.

  Unabhängig davon, ob ein Verstecken stattfindet oder nicht, hat dies aber zunächst keine weiteren Auswirkungen auf die Nutzung des Systems.

  Dieses Kriterium ist demnach vorerst nicht weiter relevant für die weitere Entwicklung des Systems.

- **Quellentransparenz:**

  Die zentrale Frage bei dieser Form der Transparenz ist, ob das System dem Benutzer verstecken soll, welche Datenquellen im Einsatz sind.

  Für den Benutzer ist es dabei durchaus eine wichtige Information, zu wissen aus welchen Quellsystemen Daten ursprünglich gekommen sind.

  Zusätzlich hat eine Einschränkung der Quellentransparenz Auswirkungen auf das mögliche Ausmaß der Datenfusion zur Folge.

  Deswegen sollen durch diese Form der Transparenz vorerst keine weiteren Einschränkungen für die Entwicklung des Systems auferlegt werden und ggf. eine niedrige Quellentransparenz angestrebt werden.

- **Schnittstellentransparenz:**

  Die Schnittstellen und Methoden um auf die Datenquellen zuzugreifen sollen komplett vom System gekapselt werden und damit gänzlich vor dem Benutzer versteckt werden.

  Eine hohe Schnittstellentransparenz ist damit strikt erforderlich und soll bei der Entwicklung des Systems berücksichtigt werden.

- **Schematransparenz:**

  Der Benutzer soll kein Wissen über die unterliegenden Datenmodelle der Datenquellen besitzen.

  Das System muss diese also gänzlich vor dem Benutzer verstecken und erfordert demnach eine hohe Schematransparenz.

Die einzelnen Entscheidungen sind in **Tabelle 2** veranschaulicht zusammengefasst.

	-	o	+	++
Ortstransparenz		X		
Quellentransparenz	X			
Schnittstellentransparenz				X
Schematransparenz				X

Tabelle 2: Erforderliche Ausprägung der einzelnen Transparenzkriterien.
Legende: (-) niedrig, (o) neutral, (+) hoch, (++) sehr hoch

Virtualisierung/Materialisierung der Daten:

An dieser Stelle werden die Vor- und Nachteile einer materialisierte oder virtuellen Integration für das System aufgestellt, sowie eine Entscheidung getroffen, welche Strategie für die weitere Entwicklung verwendet werden soll.

- **Materialisierte Integration:**

  Bei einer materielle Integration, würden zunächst alle vorhandenen Daten aus den Quellsystemen in eine zentrale Datenbank kopiert werden. Dabei findet eine Transformation

der Daten aus den lokalen Datenmodellen in das globale Datenmodell der zentralen Datenbank statt.

Alle Anfragen an das System würden dann mit diesen Kopien beantwortet werden. Je nach Synchronisationsplan, könnte nicht gewährleistet werden, dass es sich dabei immer um die aktuellsten Daten handelt.

Da über das System auch Schreibzugriffe in die jeweiligen Quellsysteme möglich sein sollen, würde zusätzlich eine Synchronisation in beide Richtung benötigt werden. Dabei könnten Konflikte bei der Synchronisation zwischen den globalen und den lokalen Daten nicht ausgeschlossen werden, und müssten gesondert gehandhabt werden.

- **Virtuelle Integration:**

  Bei einer virtuellen Integration würden die Daten in den jeweiligen Quellsystemen verbleiben. Erst bei einer Anfrage, würden die Daten direkt aus dem Quellsystem geholt und in das globale Datenmodell transformiert werden. Diese Daten hätten dadurch immer den aktuellsten Stand, da sie direkt aus dem Quellsystem selbst stammen. Konflikte bei der Synchronisation würden so vermieden werden.

  Durch die verteilte Natur der Anfragen, wird eine komplexere Anfragelogik benötigt. Zusätzlich fallen Datenzugriffe dabei ggf. in mehrere Datenquellen ab, was zu einer höheren Bearbeitungsdauer der Anfragen führen könnte.

Eine zusammengefasste Gegenüberstellung der einzelnen Vor- und Nachteile ist in **Tabelle 3** gegeben. Demnach ist eine virtuelle Integration zu bevorzugen, da dadurch Nachteile bei der Aktualität der Daten und die Gefahr von Synchronisationskonflikten entfällt.

	Materialisierung	**Virtualisierung**
Aktualität der Daten	nicht gewährleistet	immer aktuell
Anfragebearbeitung	simpel	komplex
Konflikte durch Schreibzugriffe	möglich	keine

Tabelle 3: Gegenüberstellung der Vor- und Nachteile von materialisierter und virtueller Integration.

3.3.2 Überbrückung der Heterogenitäten

Da keine Möglichkeiten zur Einschränkung der Autonomie der Datenquellen besteht, muss SPIDER in der Lage sein mit verschiedensten Ausprägungen von Heterogenitäten der Quellsysteme zurecht zu kommen.

Im Folgenden werden Ansätze vorgestellt, wie diese einzelnen Arten von Heterogenitäten im System bewältigt werden können.

- **Technische Heterogenität:**

  Jede anzubindende Datenquelle besitzt unterschiedliche Möglichkeiten um diese anzusprechen.

  Da es gefordert ist, dass alle Datenzugriffe in die Quellsysteme über ein einheitliches Interface stattfinden soll (vgl. UC 2.2), ist ein Auflösen der Heterogenität für jedes anzubindende Quellsystem notwendig.

  Ein bereits in dieser Arbeit vorgestelltes Konzept, für die Überbrückung von technischer Heterogenität bei der Anbindung eines Quellsystems, sind Wrapper.

  Bei diesem Ansatz würde für jedes anzubindende Quellsystem ein Wrapper erstellt werden, welcher die proprietären Logiken und Aufrufe des Quellsystems kapselt. Die Funktionalitäten der Quellsysteme würden damit alle unter dem festgelegten Interface der Wrapper vereint werden.

- **Syntaktische Heterogenität:**

  Für Konvertierungen zwischen verschiedenen Datenformaten wie Zeichenkodierungen bei Strings oder Datenformaten existieren bereits mehrere Lösungen, weswegen für diese Problematik keine weiteren Ansätze in dieser Arbeit untersucht werden sollen.

  Zusätzlich sind die Methoden zur Transformierung zwischen einzelnen Datenformaten stark situationsabhängig, und erfordern je nachdem eine gesonderte Fallanalyse.

- **Datenmodell Heterogenität**

  Um einem Benutzer die Daten der einzelnen Quellsysteme in einem einheitlichen Format zu präsentieren (vgl. UC2.2), muss eine Transformation des jeweiligen lokalen Datenmodells des Quellsystems in ein zentrales Datenmodell von SPIDER stattfinden.

  Dies ist ebenfalls eine Aufgabe, die von Wrappern gelöst werden kann.

- **Strukturelle Heterogenität**

  Auch wenn zwei Datenquellen das gleiche Datenmodell verwenden sollten, kann es immer noch Unterschiede innerhalb ihrer Modellierung geben.

  Aus diesem Grund fällt in diese Kategorie die gleiche Überlegung wie bei der Datenmodell Heterogenität und wird ebenfalls durch den Einsatz von Wrappern gelöst.

- **Semantische Heterogenität**

  Eine feste Anforderung an das System ist der Erhalt der Daten in einem einheitlichen Format (vgl. UC 2.2). Um dies zu gewährleisten müssen die jeweiligen Semantiken der Quellsysteme auf entsprechende Semantiken innerhalb eines globalen Datenmodells übertragen werden. Dies umfasst ggf. auch die Datenfusion von semantisch gleichen Objekten, die sich in verschiedenen Datenquellen befinden.

  Für diese Aufgabe ist man auf das Wissen von Domänenexperten angewiesen. Demnach fällt dies in den Bereich der Betreiber des Systems.

  Eine in dieser Arbeit vorgestellte Möglichkeiten für die semantische Integration ist der Einsatz von Ontologien für eine ontologiebasierte Integration der Datenquellen.

Heterogenität	Technik zur Überbrückung
Technisch	Wrapper
Syntaktisch	situationsbedingt
Datenmodell	Wrapper
Schematisch	Wrapper
Semantisch	Ontologien

Tabelle 4: Eingesetzte Techniken zur Überbrückung der einzelnen Heterogenitäten

Eine Zusammenfassung der Techniken die für die Überbrückung der einzelnen Heterogenitäten angewendet werden, ist in **Tabelle 4** zu sehen. Die genaue Umsetzung der jeweiligen Techniken soll in den nächsten Kapiteln erarbeitet und beschrieben werden.

3.3.3 Semantische Integration

In diesem Abschnitt sollen Entscheidungen für die semantische Integration im SPIDER besprochen werden.

Da bereits im vorherigen Abschnitt entschieden wurde, dass für die Überbrückung der semantischen Heterogenität auf eine ontologiebasierte Integration zurückgegriffen wird, ist es erforderlich, dass ein globales Datenmodell erstellt wird. Um die Anforderungen an dieses globale Datenmodell aufzustellen, sollen zunächst einige Betrachtungen über die Daten die

Da die Datenquellen vor der Integration nicht bekannt sind (vgl. UC1.1, UC1.2), muss das Datenmodell demnach flexibel genug sein, um möglichst viele Sachverhalte ausreichend detailliert abbilden zu können.

Da es sich bei den zu integrierenden Systemen hauptsächlich um Systeme handelt, welche im PLM Umfeld zum Einsatz kommen, sind die Systemtypen und deren verwalteten Daten bekannt. Eine Übersicht dieser Systemtypen und ihrer verwalteten Daten ist in **Tabelle 5** zu sehen.

Systemtyp	Verwaltete Daten
ALM	Anforderungen, Dokumente
PDM	Bauteile, Dokumente, Entwicklungs-Stücklisten
CASE	Software-Artefakte
E-CAD	Schaltpläne
ERP	Produktstammdaten, Produktions-Stücklisten

Tabelle 5: Im PLM Umfeld eingesetzte Systemtypen und deren verwaltete Daten

Diese Objekte sollen hauptsächlich weiterhin in ihren ursprünglichen Systemen verwaltet werden. Die für die Integration relevanten können sich dabei aus den Stammdaten oder den Metadaten zusammensetzen.

Um demnach eine lose Kopplung zwischen dem globalen Datenmodell und den eingesetzten Datenquellen zu erreichen, ist ein *Top-down-Entwurf* des globalen Datenmodells anzustreben. [24]

Zusammengefasst muss das Datenmodell in der Lage sein, die jeweiligen gewünschten Daten aus den Quellsystemen abzubilden. Da die Datenquellen sowie die Daten die für die Integration berücksichtigt werden sollen, vorher nicht bekannt sind, muss demnach bei der Einführung des Systems das globale Datenmodell auf die spezifischen Anforderungen des Unternehmens, sowie auf die eingesetzten Datenquellen, zugeschnitten werden.

Ebenso muss es möglich sein Änderungen am eingesetzten Datenmodell durchführen zu können, um auf mögliche Änderungen in der Systemlandschaft reagieren zu können. Eine in dieser Arbeit vorgestellte Methode zur Beschreibung von semantischen Datenmodellen ist RDF. Bei einer Verwendung von RDF könnte das Datenmodell zunächst unabhängig vom eingesetzten System selbst geschrieben werden. Diese Trennung von Datenmodell und System ist in diesem Fall hilfreich, da SPIDER damit nur in der Lage sein muss, die Semantiken in Form von RDF zu verstehen, und die jeweiligen Gegebenheiten der eingesetzten Systemlandschaft dabei nur in Form des globalen RDF Datenmodells ausgedrückt werden müssen.

Aus diesem Grund wird für SPIDER eine Top-Down Ontologie entwickelt, welche für die Beschreibung der einzelnen Datenquellen verwendet werden kann. Die Entwicklung dieser Ontologie soll in einem der folgenden Kapitel weiter betrachtet werden.

Da RDF nur eine Spezifikation vorgibt, muss eine konkrete Technologie gewählt werden, welche diesen Standard umsetzt. Da dabei alle Implementierungen, welche die Spezifikation erfüllen, gleich mächtig sind, spielt es keine Rolle welche tatsächlich gewählt wird. Aus persönlicher Präferenz wurde demnach die Entscheidung auf JSON-LD gesetzt.

JSON-LD ist eine leichtgewichtige Implementierung des RDF Standards welches auf dem JSON Datenformat basiert. [1] Das JSON Datenformat selbst ist ein simples Format zur Serialisierung von Daten, das auf einem Key-Value-Prinzip basiert. [22]

JSON-LD wird verwendet für den Austausch von Informationen. Einer der Vorteile von JSON-LD ist, dass Anwendungen für die Konsumierung von JSON-LD Daten nicht auf eine komplette Implementierung einer RDF-Engine angewiesen sind, sondern es ausreicht in der Lage zu sein Daten im JSON Format verarbeiten zu können. [1]

Das Datenmodell von JSON-LD bildet einen gerichteten, markierten Graphen. Jede Node im Graphen entspricht einer RDF Ressource und ist markiert mit einem *Internationalized Resource Identifier* (IRI) mit welchem diese eindeutig identifizierbar ist. Dieses Element besitzt im JSON-LD Dokument den Schlüssel *@id* und sein Wert entspricht dem IRI. [1]

Eine Node kann mehrere RDF Properties besitzen. Dabei werden diese als einfache JSON Werte in der Node festgehalten. Da RDF Properties auch gleichzeitig Ressourcen sind, besitzen diese ebenfalls einen IRI. Der IRI der Property wird dabei als Key-Element für die Serialisierung

einer Node verwendet, mit dem entsprechenden Wert den diese Node für diese Property hat, als Value. [1]

Damit Properties nicht jedesmal aufs Neue definiert werden müssen, bietet JSON-LD das *@context* Element an. In diesem lassen sich IRIs aus externen Quellen an einen Bezeichner binden, welcher dann im weiteren Dokument verwendet werden kann. Auf diese Weise lassen sich Vokabulare aufbauen, auf welche dann beliebig in JSON-LD zurückgegriffen werden kann. [1]

3.3.4 Semantische Beschreibung der Datenquellen

Damit eine eine Datenquelle an SPIDER angeschlossen werden kann, muss diese zunächst mittels RDF semantisch beschrieben werden. Das Erstellen dieser Beschreibungen ist die Aufgabe der Betreiber von SPIDER. Diese müssen als Domänenexperten entscheiden, welche Daten die anzuschließende Datenquelle zur Verfügung stellen soll. Anschließend muss ein Abbild dieses Datenmodells in Form eines JSON-LD Dokumentes erzeugt werden.

Dabei wird auf eine Objekt-orientierte Modellierung zurückgegriffen. Für jeden Artikel den die entsprechende Datenquelle der Integration zur Verfügung stellen will, muss zunächst eine Klasse definiert werden. Dabei wird im JSON-LD Dokument eine neue Node vom Typen **<rdf:class>** erstellt. Zusätzlich wird dieser Node eine IRI zugewiesen, durch welche man diese eindeutig im Netzwerk erreichen kann, sowie ein Label, welches den lesbaren Namen dieser Klasse wiedergibt und ein Kommentar, zu Dokumentationszwecken.

In **Listing 1** ist ein JSON-LD Dokument zu sehen, welches die Definition einer Klasse für ein generisches Produkt, enthält.

Listing 1: Beispiel einer Klassendefinition für ein Produkt innerhalb eines JSON-LD Dokuments.

```
{
  "@context": {
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "product": "http://www.example.org/plm/schema/Product.jsonld#",
  },
  "@graph": [
        {
                "@id": "part",
                "@type": "rdfs:class",
                "rdfs:comment": "A simple product.",
                "rdfs:label": "Product",
    }
  ]
}
```

Diesen Klassen werden dann entsprechende Attribute hinzugefügt, welche die Stammdaten des jeweiligen Artikels abbilden. Bei der Wahl der Attribute kann frei entschieden werden, welche Daten man zur Verfügung stellen will. So kann es bspw. sinnvoll sein, nur ein Subset der

Stammdaten in das Datenmodell übertragen, wenn die restlichen Daten für die gewünschten Prozessabläufe nicht von Interesse sind. Auch kann man zusätzliche Metadaten zur Verfügung stellen, wie bspw. Versionsnummern oder Änderungsindizes.

Für die Deklarierung eines Attributes wird eine neue JSON-LD Node angelegt mit dem Typen *rdf:property*. Diese wird, wie bei der Definition einer Klasse, mit den zusätzlichen Properties für Label und Kommentar ergänzt.

Die Zuordnung des Attributs zu einer Klasse erfolgt über das aus der Schema.org-Ontologie[6] stammende Property *schema:domainIncludes*. In dieses wird die Node-Id der jeweiligen Klasse eingefügt.

Zusätzlich muss ein Datentyp für dieses Attribut festgelegt werden. Dabei kann auf Datentypen aus vorhandenen Vokabularen zurückgegriffen werden, wie bspw. **rdfs:text** zur Beschreibung von textuellen Daten oder **<xmls:date>** für ein Datum. Anwender können anhand dieses Wertes entnehmen, wie dieses Attribut weiter zu verarbeiten ist.

Eine JSON-LD Node für die Deklarierung eines Attributes ist in**Listing 2** zu sehen.

Listing 2: Deklarierung eines Attributes für eine Klasse innerhalb einer JSON-LD Node

```
{
  "@id": "product:delivery_date",
  "@type": "rdf:Property",
  "rdfs:comment": "Product delivery date.",
  "rdfs:label": "Delivery Date",
  "schema:domainIncludes": [
    {
      "@id": "product"
    }
  ],
  "schema:rangeIncludes": [
    {
      "@id": "xmls:date"
    }
  ]
}
```

3.4 Erstellen der SPIDER-Ontologie

In diesem Abschnitt wird die SPIDER-Ontologie erarbeitet, welche als Top-down Ontologie für die Beschreibung der anzubindenden Quellsysteme verwendet werden soll. Dafür wird zunächst eine Betrachtung ausgewählter Systeme, die im PLM Umfeld eingesetzt werden, vorgenommen. Aus diesen Beobachtungen wird anschließend abgeleitet, welche semantischen Eigenschaften in die SPIDER-Ontologie übernommen werden müssen.

3.4.1 Betrachtung ausgewählter Quellsysteme

- **Aras Innovator**

  Aras Innovator ist ein von der Firma Aras entwickeltes PLM System. Es integriert dabei die Funktionalität eines PDM Systems und ermöglicht die Verwaltung von Produkten Produktstrukturen u. Dokumenten, sowie Versionskontrolle und Änderungsmanagement.

  Aras Innovator wird über eine Web-Oberfläche bedient, welche auf ein zentrales Repository zurückgreift, in dem sich die Daten befinden. Dieses Repository lässt sich ebenfalls über eine Web-Schnittstelle ansprechen. Ein wichtiges Feature von Aras Innovator ist die Möglichkeit über eingebaute Werkzeuge ein Customizing des Systems vorzunehmen. So bietet das System Möglichkeiten zur Erstellung eines eigenen Datenmodells, dem Gestalten von eigenen Eingabemasken und Bedienoberflächen, sowie dem Verändern des Verhaltens des Systems durch das Schreiben und Einbetten von eigenen Programmteilen.

  An dieser Stelle soll eine Betrachtung der PDM Funktionalität von Aras Innovator als mögliche Datenquelle für SPIDER durchgeführt werden. Anhand dieser Fallstudie sollen Erkenntnisse gezogen werden, welche Eigenschaften das Datenmodell von Aras Innovator als PDM System besitzt, und welche Eigenschaften sich auf Datenmodelle von Quellsystemen im Allgemeinen übertragen lassen.

  In **Abb. 10** ist ein Screenshot der Eingabemaske für ein Bauteil aus Aras Innovator zu sehen. Die Maske besteht aus einzelnen Eingabefeldern, sowie Reitern zur Anzeige von weiteren Daten und Referenzen. Bei den Eingabefeldern handelt es sich um für die bei der Entwicklung relevanten Attribute, wie Typus, Gültigkeitsdatum, Stückzahlnummern, etc. Diese Attribute sind von den Ingenieuren frei veränderbar.

  Bei dem Feld in (1) handelt es sich um die Teilenummer des Bauteiles. Das Konzept der Teilenummern wurde bereits vorgestellt und dient an dieser Stelle zu einer eindeutigen Identifizierung des Bauteils innerhalb des Aras Innovator Systems. Demnach ist vom Quellsystem gewährleistet, dass es kein anderes Bauteil mit der gleichen Teilenummer geben kann.

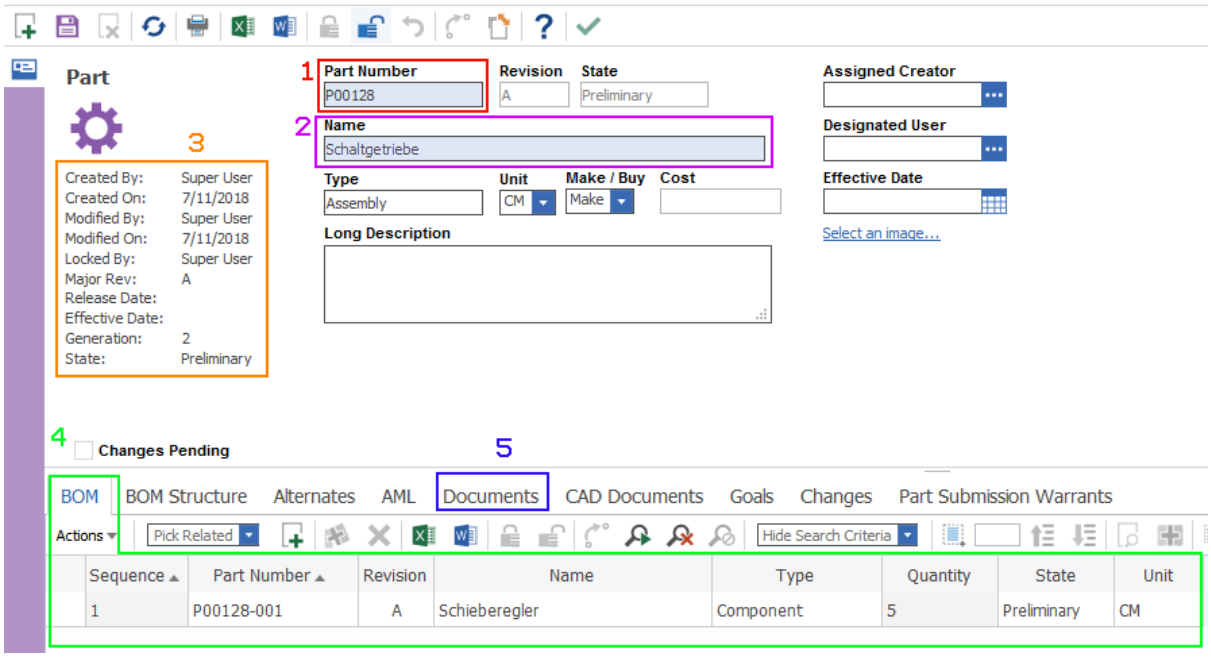

Abbildung 10: Ausschnitt einer Eingabemaske aus Aras Innovator mit Annotationen. (1) Teilenummer, (2) Pflichtfeld, (3) Systemattribute,(4) Produktstruktur, (5) Angefügte Dokumente

In Feld (2) ist ein *Pflichtfeld* zu sehen. Diese werden in der Eingabemaske farblich hellblau unterlegt. Dies sind Attribute, welche immer einen Wert haben müssen. Bei der Erstellung eines neuen Bauteils wird vom System verlangt, dass diese Werte eingetragen werden. Auch ein nachträgliches Löschen der Werte darf nicht stattfinden und wird deswegen vom Quellsystem untersagt. [13]

Den frei-änderbaren Attributen gegenüber stehen system-verwaltete Attribute (3) wie Modifikationsdatum, Version, Autor, etc. Diese Attribute werden vom Quellsystem automatisch gepflegt. Ein Auslesen dieser Attribute ist möglich aber manuelle Änderungen an diesen Werten werden vom System untersagt. [13]

In (4) ist der Reiter für die Stückliste der Baugruppe zu erkennen. In dieser befindet sich ein Verweis auf ein anderes im System existierendes Bauteil. Zu diesem Verweis ist ebenfalls die Stückzahl des Vorkommens innerhalb dieser Stückliste gespeichert.

In den weiteren Reitern bei (5) werden weitere Referenzen und Verweise zu anderen Elementen des PDM Systems angezeigt. Dort können u.a. Projektzuordnungen eingesehen werden oder angefügte Dokumente des Bauteils angezeigt oder hinzugefügt werden. Zu beachten ist, dass es auch implizite Referenzen geben kann, wenn das Bauteil beispielsweise selbst in einer anderen Baugruppe verbaut ist.

Eine mögliche Anwendung für die Integration von Aras Innovator als PDM System in SPIDER, wäre eine Verfügungstellung der Bauteile für CRUD-Zugriffe.

Diese Zugriffe müssen konsistent sein zu dem internen Datenmodell von Aras Innovator. Demnach müsste in der semantischen Beschreibung auslesbar sein, dass bestimmte Attribute, wie bspw. die System-Attribute nicht verändert werden dürfen oder das Werte für bestimmte Attribute zum Erstellen von neuen Bauteilen vorhanden sein müssen.

- **Git**

  Git ist eine freie Software für die Versionskontrolle von Quelltexten. Die Entwicklung wird von der Open-Source Community vorgenommen und es ist lizenziert unter der GNU General Public License. [19]

  Git wird hauptsächlich für das Verwalten und Verfolgen von Änderungen von Software-Quelltexten verwendet. Die Quelltexte sowie die verfolgten Änderungen werden in einem Repository gespeichert. Jede Änderung wird dabei als eine neue Revision im Repository hinterlegt. Eine komplette Historie des Repositories ist somit jederzeit nachvollziehbar. Um auf diesen Repositories zu arbeiten, wird das Kommandozeilen Programm Git verwendet. [19]

  Ein neues Repository wird mit dem Befehl *git init* erzeugt. Dabei wird ein neues Repository im aktuellen Verzeichnis des Dateisystems erzeugt. Dateien, die in dieses Verzeichnis abgelegt werden, können per *git add* in das Repository hinzugefügt werden.

  Änderungen an Dateien im Repository werden von Git verfolgt und können mittels *git commit* in das Repository eingespielt werden. Dabei wird eine neue Revision erzeugt. Jede Revision ist dabei eindeutig durch einen SHA-1 Code eindeutig identifizierbar. [19]

  Ein möglicher Anwendungsfall für die Anbindung eines Git-Repositories an SPIDER, wäre das Verfügbarmachen von Metadaten der Repositories, also bspw. Auslesen von Informationen über verschiedene Revisionen.

  Um die Integrität des Repositories zu gewährleisten sollte es dabei vermieden werden, neue Revisionen über SPIDER zu erstellen oder vorhandene zu editieren. Demnach müsste in der semantischen Beschreibung des Git-Repositories als Datenquelle auslesbar sein, dass Schreibzugriffe auf Revisionen untersagt sind.

3.4.2 Aufstellen der Ontologie

Aus den Betrachtungen der beiden Datenquellen sollen hier nun allgemeine Eigenschaften spezifiziert werden, die für die Modellierung der Datenquellen in SPIDER verwendet werden können.

Bis jetzt wurde bei der Modellierung der Datenquellen nur ein Objekt-orientiertes Abbild der einzelnen lokalen Datenmodelle vorgenommen. Daten die eine Datenquelle zur Verfügung stellen will, sind damit in der Form von Klassen mit ihren zugehörigen Attributen und Datentypen vorhanden.

Im Hinblick auf die Möglichkeit durch SPIDER auch Schreibzugriffe in die Quellsysteme zu setzen, wurde in den beiden Fallstudien gezeigt, dass eine reine Modellierung der Datenquellen nicht ausreicht, da somit bestimmte Semantiken des Quellsystems nicht berücksichtigt werden.

Um dieses Problem zu lösen, wird in dieser Arbeit eine Ontologie vorgeschlagen, um die erlaubten Operationen und Zugriffe innerhalb eines Quellsystems semantisch in dessen Datenmodell zu beschreiben.

Diese Ontologie muss zunächst erlauben auszudrücken, welche CRUD-Operationen auf einer von der Datenquelle verwalteten Klasse erlaubt sind. Dafür sind in der SPIDER-Ontologie die RDF-Properties **<spider:read>**,**<spider:create>**,**<spider:update>** und **<spider:delete>** definiert. Diese lassen sich als Properties für mittels **<rdfs:class>** definierter Klasse anwenden. Eine Auflistung und genaue Beschreibung dieser Properties ist in **Tabelle 6** zu sehen.

Eigenschaft	Beschreibung
Read	Legt fest ob Lese-Zugriffe auf die Objekte einer Klasse stattfinden dürfen.
Create	Legt fest ob neue Objekte dieser Klasse erzeugt werden dürfen.
Update	Legt fest ob Schreib-Zugriffe auf den Objekten einer Klasse stattfinden dürfen.
Delete	Legt fest ob vorhandene Objekte einer Klasse gelöscht werden dürfen.

Tabelle 6: Definierte Semantiken der SPIDER-Ontologie für Klassen.

Diese Semantiken beschreiben bis jetzt nur das Verhalten von Objekten auf Klassenebene. Um weitere Konzepte, wie bspw. das nicht erlaubte Verändern von System-Attribute zu beschreiben, müssen weitere Properties definiert werden.

Die Eigenschaften, die von diesen Properties abgedeckt werden müssen, werden im Folgenden vorgestellt:

- **Identifizierende Attribute:**

  Dies sind Attribute, welche eine exakte Identifizierung eines Objektes innerhalb einer Datenquelle ermöglichen, wie bspw. die Teilenummer eines Bauteiles im PDM System, oder der SHA-1 Code einer Revision im Git.

  Es wird im Kontext von SPIDER davon ausgegangen, dass es für jede Klasse innerhalb einer Datenquelle solch ein Attribut gibt. Das Identifizierende Attribut kann dadurch auch von SPIDER für die eindeutige Identifizierung innerhalb des globalen Systems verwendet werden. Deswegen wird eine semantische Kennzeichnung dieser Eigenschaft benötigt.

- **Benötigte Attribute zur Objekterzeugung:**

  Im Beispiel des Bauteils in Aras Innovator war zu sehen, dass für eine neue Erstellung eines Bauteils, das Ausfüllen bestimmter Felder benötigt wurde. Dieses Konzept von bestimmten Attributen, welche für die Objekterzeugung ihrer jeweiligen Klassen benötigt

werden, muss ebenfalls in der semantischen Beschreibung der Datenquelle abgebildet werden.

- **Schreibschutz bestimmter Attribute:**

  Wie am Beispiel der Systemattribute von Aras Innovator zu sehen war, gibt es Attribute, bei denen die internen Logiken der Datenquelle Änderungen verbieten.

  Demnach muss in der semantischen Beschreibung der Datenquelle erkennbar sein, welche Attribute nicht veränderbar sind.

Für diese Eigenschaften wurden in der SPIDER-Ontologie ebenfalls Properties angelegt. Die Namen dieser Properties und ihre Beschreibung sind in **Tabelle 7** zu sehen.

Eigenschaft	Beschreibung
Get	Legt fest ob ein Attribut gelesen werden kann
Set	Legt fest ob ein Property verändert werden darf
Required	Legt fest ob ein Property bei Objekterzeugung mit einem Wert vorhanden sein muss
UniqueID	Definiert das angegebene Property als eindeutigen Identifier für eine eindeutige Identifizierung eines Objektes. Dieses Property muss bei der Definition einer Klasse innerhalb der semantischen Beschreibung einer Datenquelle exakt einmal vorkommen.

Tabelle 7: Benötigte Semantiken der SPIDER-Ontologie für Attribute.

Listing 3: Some JSON code

```
{
  "@id": "spider",
  "@type": "http://www.w3.org/2002/07/owl#Ontology",
  "label": "The SPIDER Vocabulary",
  "description": "Contains annotations used by SPIDER."
},
{
  "@id": "spider:get",
  "@type": "xsd:boolean",
  "label": "Get",
  "description": "Defines if a property should be returned in a SPIDER Get accesses
      .",
  "isDefinedBy": "spider"
},
```

3.5 Architektur

In diesem Abschnitt soll, ausgehend von den konkreten Anforderungen der Use Cases und den Anforderungen der Integration, die Architektur für SPIDER erarbeitet werden. Dafür werden zunächst die einzelnen Komponenten des Systems identifiziert.

3.5.1 Komponenten

Die hier aufgelisteten Entitäten sind Komponenten die für den Einsatz im System identifiziert wurden.

- **SPIDER-Core:**

  Es wurde bereits festgestellt, dass für die Verwaltung der angebundenen Datenquellen sowie für die Speicherung der Zusatzinformationen ein Backbone existieren muss, welches außerhalb der angebundenen Datenquellen selbst agiert.

  An diesem Backbone sollen die Datenquellen sowie deren Datenmodelle registriert werden können. Ebenso sollen hier die Zusatzinformationen zu den verschiedenen von den Datenquellen verwalteten Objekten abgefragt werden können.

- **Wrapper:**

  Für die Überbrückung der technischen, Datenmodell und schematischen Heterogenitäten der Datenquellen wurde bereits festgelegt, auf die Verwendung von Wrappern zurückzugreifen. Demnach muss für jede anzubindende Datenquelle ein Wrapper erstellt werden, der die internen und proprietären Logiken der Datenquelle kapselt. Diese Wrapper besitzen alle ein einheitliches Interface für die Kommunikation zur Aussenwelt.

- **Datenquellen:**

  Dies sind die eingesetzten IT-Applikationen des Unternehmens in dem SPIDER eingesetzt wird.

3.5.2 Service-orientierte Architektur

Stattdessen ist eine Service-orientierte Architektur (SOA) ein weit-verbreitetes Lösungskonzept für den Einsatz verteilter Softwarekomponenten. Bei dieser wird Funktionalität in sog. Services gekapselt. Diese Services werden meist unter Zuhilfenahme von Web-Technologien implementiert und anschließend zusammen mit Informationen über ihre Interfaces in einem Netzwerk oder dem Internet veröffentlicht. [33]

Das Konzept einer traditionellen SOA ist in **Abb. 11** zu sehen. Dort wird zwischen drei verschiedenen Komponenten unterschieden:

- **Service-Provider:**

  Service-Provider implementieren Services und stellen diese der Außenwelt zur Verfügung, indem Informationen über ihre Schnittstellen und den Service selbst an einem Service-Broker registriert werden.

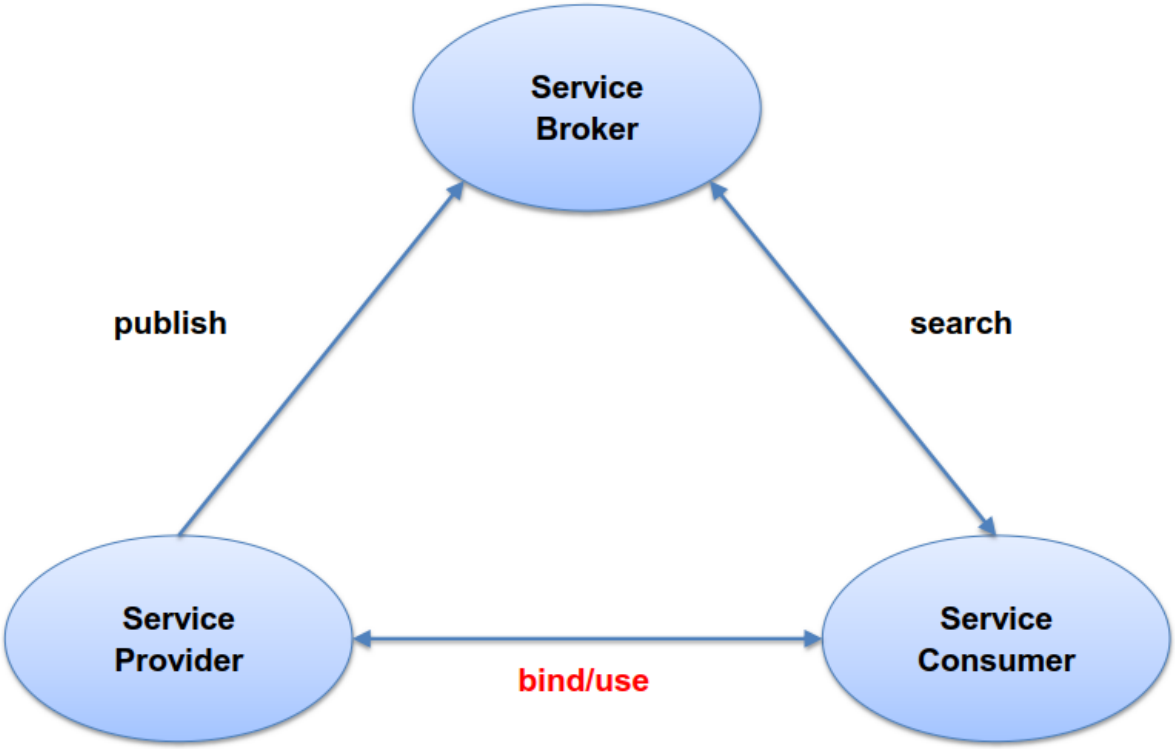

Abbildung 11: Aufbau einer Service-orientierten Architektur Quelle: [30]

- **Service-Broker:**

  Service-Broker erlauben es Service-Providern ihre Services zu registrieren und speichern deren Meta-Informationen. Service-Consumer können Informationen über registrierte Services am Broker anfordern.

- **Service-Consumer:**

  Consumer, welche einen Service in Anspruch nehmen wollen, erkundigen sich zunächst an einem Broker über vorhandene Services. Wurde ein passender Service-Anbieter lokalisiert, können diese daraufhin über Standard-Protokolle mit diesem Service interagieren.

Die Komponenten von SPIDER lassen sich demnach auf die Konzepte einer SOA übertragen. Die Wrapper wären in diesem Sinne Service Provider, die in Form von Datenintegration einen Service zur Verfügung stellen. Der SPIDER-Core würde als Service Broker fungieren, an dem sich die Wrapper mit den Informationen über ihre Datenquelle registrieren. Die Benutzer des Systems würden in der Form von Service Consumern den SPIDER-Core zunächst nach Informationen über die angebundenen Datenquellen durchsuchen und anschließend ihre Anfragen an den jeweiligen Wrapper der Datenquelle senden.

3.5.3 Betrachtete Alternativen

Ein monolithischer Aufbau des Systems, in dem die Wrapper und der Core alle Bestandteil eines großen Software-Systems sind, wäre zunächst eine Möglichkeit gewesen das System zu

entwickeln, aber bei genauerer Betrachtung war festzustellen, dass diese Architektur für die Zwecke des SPIDER nicht ausreichend wäre.
Die Gründe dafür seien der folgenden Liste zu entnehmen:

- **Kopplung:**

  Jede Anbindung eines Quellsystems als festen Bestandteil des monolithischen Systems zu implementieren, würde eine hohe Kopplung der beteiligten Quellsysteme an das System zur Folge haben.

- **Wartbarkeit:**

  Änderungen am Wrapper einer Datenquelle, hätten zur Folge, dass das gesamte monolithische System verändert werden müsste.

3.6 Kommunikation

Für die Architektur von SPIDER wird auf eine SOA gesetzt. Demnach werden der SPIDER-Core und die einzelnen Wrapper in der Form von Web-Services implementiert und die Kommunikation innerhalb sowie außerhalb des Systems soll mittels Web-Technologien stattfinden. In diesem Kapitel soll festgelegt werden, welche konkreten Technologien für die Kommunikation und Implementierung der Web-Services verwendet werden. Dabei werden zunächst mit SOAP und REST zwei gängige Technologien für die Implementierung von Web-Services vorgestellt und verglichen.

3.6.1 Evaluierung von SOAP

SOAP ist eine Empfehlung des *World Wide Web Consortium* (W3C). Es beschreibt die Spezifikation für ein leichtgewichtiges Protokoll für den Datenaustausch von Web-Services in einer dezentralen, verteilten Umgebung unter der Verwendung von XML.

Der Hauptgedanke von SOAP ist der Austausch von *SOAP Nachrichten* zwischen sog. *SOAP Nodes*. Als SOAP Node werden dabei jene Anwendungen bezeichnet, welche die benötigte Verarbeitungslogik umsetzen, um SOAP Nachrichten zu versenden, zu empfangen oder verarbeiten zu können.

Eine SOAP Nachricht besteht zunächst aus dem SOAP Envelope, welcher den gesamten Container der SOAP Nachricht ausmacht. In diesem enthalten sind der Kopfteil der Nachricht, der sog. SOAP Header und der SOAP Body. Der SOAP Header enthält optionale Metadaten für die Verarbeitung der Nachricht, während im SOAP Body anwendungsspezifische Daten enthalten sind.

Der Austausch der Nachrichten erfolgt über Standard HTTP-Methoden. Dabei bietet SOAP Unterstützung für verschiedene Message Exchange Patterns, also Strategien des Nachrichtenaustausches. Die verbreitetste Variante davon ist, das Fire-and-forget Pattern. Mit diesem werden Nachrichten zwischen Anwendungen ausgetauscht, ohne dass von der Spezifikation eine bestimmte Antwort verlangt wird. Die Reihenfolge und Form der Nachrichten ist damit der Anwendung selbst überlassen. Dies führt zu einer asynchronen Kommunikation und der Entkopplung von Kommunikationspartnern. Die SOAP Nachricht fungiert damit letztlich als Wrapper für den XML Inhalt der Nachricht. Aufrufe erfolgen dabei an einen vorher bestimmten Endpunkt des Services und verwenden ausschließlich die Standard HTTP-Methode *POST*. Jegliches weitere Verhalten der Anwendung wird bestimmt durch die Verarbeitung des XML Inhalts der Nachricht.

3.6.2 Evaluierung von REST

Im Gegensatz zu SOAP ist REST keine reine Spezifikation, sondern ein architektureller Stil für die Implementierung von Web-Services. REST definiert eine Reihe von losen Regeln, welche eine Anwendung erfüllen muss, damit diese als *RESTful Service* gelten kann.

So wird in den Regeln festgelegt, dass die Kommunikation zwischen Service und Anwender zustandslos ablaufen soll. Das heißt, jede Anfrage enthält alle benötigten Informationen um die Anfrage verarbeiten zu können und ist nicht von vorherigen Anfragen abhängig. Dies ermöglicht eine parallelisierte Verarbeitung verschiedener Anfragen und erhöht die Skalierbarkeit des Services.

Ist es gewünscht, dass ein Zustand zwischen Anwendungen existiert, lässt sich dies dadurch simulieren, in dem der innerhalb der Anfrage kodiert wird und somit auf der Seite des Clients gespeichert wird.

Ein weiteres wichtiges Konzept von REST sind *Resourcen*. So soll jedes identifizierbare Objekt eines Services von einer Resource repräsentiert werden. Diese werden eindeutig über den *Resource Identifier* in Form einer URL angesprochen. Dies können beispielhaft einzelne Objekte oder aber auch Sammlungen von Objekten (Collections) sein. **Tabelle 8** zeigt wie Resource Identifier für diese genannten Fälle aussehen könnten.

Art der Resource	IRI
Einzelnes Objekt	http://www.example.org/Objects/ID101
Sammlung	http://www.example.org/Objects

Tabelle 8: Beispiele für verschiedene Arten einer IRI von REST-Resourcen

Eine Eigenschaft der Resource Identifier ist, dass diese global eindeutig sind. Auf diese Resourcen kann mittels Standard HTTP-Methoden zugegriffen werden. Dabei ist die Semantik des Zugriffs bereits fest von der Methode abhängig vorgeschrieben, wie in **Tabelle 9** zu sehen ist.

Methode	Semantik
GET	Fordert eine Repräsentation der Resource an.
HEAD	Identisch wie GET, aber fordert keinen Nachrichten-Body an.
POST	Fordert ein Update der Resource mit gegebenen Werten.
PUT	Fordert die Erstellung einer neuen Resource mit gegebeen Werten.
DELETE	Fordert die Löschung der Resource an.
OPTION	Fordert eine Liste der vom Server zur Verfügung gestellten Methoden an.

Tabelle 9: HTTP-Methoden und deren zugewiesene Semantiken in REST

Zusätzlich erfolgt eine strikte Trennung zwischen den Daten selbst und ihrer Repräsentation. Hierfür wird das *Content-Type* Feld des HTTP-Headers der Anfrage verwendet. So kann eine Anwendung eine für den Menschen lesbare Darstellung angefordert werden, wenn die Resource beispielsweise mit einem Web-Browser angefordert wird. Anwendungen könnten aber auch ein maschinell-verarbeitbares Format anfordern. Dabei ist zu beachten, dass es den Servern selbst überlassen ist, welche Formate diese zur Verfügung stellen, und diese auch letztendlich serverseitig implementiert sein müssen.

3.6.3 Gegenüberstellung

Zunächst wären für die Implementierung des SPIDER beide Protokolle einsetzbar. Eine genauere Betrachtung lässt jedoch REST als die vorteilhaftere Variante des Datenaustausches für den Zweck von SPIDER erscheinen.

Da in REST die Semantik bereits innerhalb des Zugriffs selbst kodiert ist (anhand der entsprechenden Methode), muss dies nicht weiter innerhalb der Nachricht selbst spezifiziert werden. Dies führt zu einer Vereinfachung des Datenformats. Zusätzlich lässt sich das Konzept der Objekte und Collections als Resourcen von REST gut auf das in RDF verwendete gleichnamige Konzept der Resourcen übertragen. Dies hätte den Vorteil, dass das in SPIDER eingesetzte Datenmodell somit gleichzeitig als Teil des REST Interfaces fungieren würde. Jedes von einer Datenquelle zur Verfügung gestellte Objekt stellt damit eine Resource in der REST-API des jeweiligen Wrappers dar. Als weiterer Vorteil, werden Antworten auf REST Anfragen bereits in den Nachrichten-Body der HTTP-Response geschrieben, und müssen nicht erst in einer gesonderten Nachricht versendet werden, wie es in SOAP der Fall wäre.

Dies führt zu einer Vereinfachung der Kommunikation zwischen SPIDER und möglichen Nutzern, da diese lediglich Standard HTTP-Methoden anwenden müssen, und nicht erst das Protokoll mittels SOAP implementieren müssen.

3.7 SPIDER-Core

Der SPIDER-Core stellt das zentrale Element von SPIDER dar. Dieser ist zuständig für die Verwaltung der angebundenen Datenquellen, sowie des föderierten globalen Datenmodells in SPIDER.

Dafür muss der SPIDER-Core den Betreibern Möglichkeiten zur Verfügung stellen, um Datenquellen möglichst unkompliziert und ohne grössere Aufwände am System zu registrieren. Anwender des Systems sollen dabei über Web-Interfaces Informationen über die angebunden Datenquellen sowie ihrer Datenmodelle zu erhalten.

3.7.1 Registrierung von Quellsystemen

Die Betreiber des Systems sind für die Verwaltung der Datenquellen, die an SPIDER angebunden werden, zuständig. Um diese Aufgabe zu erleichtern, soll der SPIDER-Core Möglichkeiten zur Verfügung stellen, um diese Verwaltung an einer zentralen Stelle vornehmen zu können. Dazu gehört das Eintragen der Informationen über eine Datenquelle. Dies umfasst:

- Den Namen der Datenquelle
- Der im Netzwerk erreichbare Endpunkt des Wrapper-Services der Datenquelle
- Das Datenmodell der Datenquelle

Das Datenmodell einer Datenquelle besteht aus den semantischen Beschreibungen ihrer zur Verfügung gestellten Objekte. Diese sollen anhand von JSON-LD Schema Dokumenten beschrieben werden. Dabei wird eine Objekt-orientierte Modellierung dieser Objekte vorgenommen indem Schemadefinitionen für Klassen sowie deren Attribute erstellt werden. Um auch die erlaubten Operationen auf diesen Objekten zu beschreiben, wurde in dieser Arbeit bereits die SPIDER-Ontologie vorgestellt. Diese einzelnen Schemadefinitionen sollen innerhalb des SPIDER-Core gesammelt werden. Dadurch wird eine Wiederverwendung einzelner Elemente für die Modellierung ermöglicht.

Um die Verwaltung der Datenquellen zu erleichtern, soll für die Betreiber eine Benutzer-Oberfläche zur Verfügung gestellt werden, an dem sich die Konfigurationen der Datenquelle durchführen lassen. Ebenfalls sollen Tools für eine Modellierung der Schemadefinitionen in dieser Oberfläche verfügbar sein. Der Betreiber soll dabei in der Oberfläche die Möglichkeit haben Klassen und Attribute zu definieren, den Klassen vorhandene Attribute zuweisen zu können, sowie einer Datenquelle selbst die definierten Klassen zuordnen zu können.

3.7.2 Web-Interfaces

In diesem Abschnitt befindet sich die Spezifikation des Web-Interfaces von SPIDER-Core.

Dieser soll zusätzlich zu einer REST-Schnittstelle auch eine simple HTTP-Schnittstelle anbieten, auf welcher die gespeicherten JSON-LD Schemadefinition abrufbar sind.

- **REST-Interface:**

  Das REST-Interface des SPIDER-Core besteht nur aus der Resource *SourceSystem*. Diese REST-Resource ist eine Collection aller am SPIDER-Core registrierten Datenquellen. Das Interface besteht dabei nur aus der HTTP-Methode GET und ist in **Tabelle 10** zu entnehmen.

Methode	Beschreibung
GET	Liefert eine Liste mit allen registrierten Datenquellen zurück. Diese enthält den Namen, den Endpunkt sowie die Pfade zur Schemadefinition der von der Datenquelle zur Verfügung gestellten Objekte.

Tabelle 10: Spezifikationen der REST-API für die Resource SourceSystem des SPIDER-Core.

- **HTTP-Schnittstelle:**

  Um ein JSON-LD Dokument verarbeiten zu können, muss dessen Schemadefinition in Form eines JSON-LD Dokumentes frei im Internet abrufbar sein. Aus diesem Grund soll der SPIDER-Core eine HTTP-Schnittstelle anbieten, an welcher die in der Konfiguration erstellten Schemadefinitionen erreichbar sind.

3.7.3 Einsatz der Aras Innovator Middleware

Für die Realisierung des SPIDER-Core wäre zunächst eine eigene Implementierung denkbar. Diese müsste konkret folgende Anforderungen erfüllen:

- Bereitstellung einer REST-Schnittstelle
- Bereitstellung einer Benutzer-Oberfläche
- Speicherung der Informationen über Datenquellen
- Speicherung der Schemadefinitionen

Da eine eigene Implementierung einen erheblichen Aufwand mit sich bringen würde, wurde stattdessen nach Middlewares gesucht, welche sich für einen Einsatz eignen würden. Durch ihre Nähe zum Produktlebenszyklus wurden dabei speziell zunächst vorhandene PLM Lösungen betrachtet.

Eine PLM Lösung, die in dieser Arbeit bereits vorgestellt wurde, ist Aras Innovator. Eine Aras Innovator Installation stellt zwei Komponenten zur Verfügung:

- Ein Backbone in dem von Aras Innovator verwaltete Daten gespeichert werden
- Eine Web-Oberfläche über welche Daten eingegeben werden können.

Als im PLM Umfeld eingesetzte Middleware, ist es möglich große Teile der Software zu modifizieren. Dies umfasst sowohl das proprietäre interne Datenmodell als auch die Eingabemasken innerhalb der Web-Oberfläche. Alle Änderungen lassen sich dabei über zur Verfügung gestellte Funktionen innerhalb der Web-Oberfläche vornehmen. [13]

Es ließe sich demnach, mit den von Aras Innovator zur Verfügung gestellten Möglichkeiten, eine Verwaltung für die Datenquellen sowie Modellierungstools für die Schemadefinitionen erstellen. Damit wären die Anforderungen der Benutzer-Oberfläche sowie der Speicherung der Informationen erfüllt. Nachträglich müsste nur noch ein REST-Interface zur Verfügung gestellt werden, um die Spezifikationen des SPIDER-Core zu erfüllen.

3.8 Wrapper

In diesem Kapitel soll das Design für den in SPIDER eingesetzten Wrapper erarbeitet werden. Das Hauptproblem dabei ist, dass es sich bei den Wrappern jeweils um individuelle Software-Komponenten handelt, welche an ihre jeweilige Datenquelle sowie das verwendete Datenmodell angepasst werden müssen.

Darum wird zunächst versucht einen generischen Wrapper als sog. *Core Asset* zu definieren. Core Assets sind bestimmte Software-Komponenten, welche im Hinblick auf die Wiederverewndung erstellt wurden. Dafür wird zunächst untersucht welche Teile des Wrappers sich nicht voneinander unterscheiden (Commonality). Anschließend wird versucht die Varianz zu identifizieren. (Variance) [18]

3.8.1 REST-Interfaces

In diesem Abschnitt wird die Spezifikation für das generische REST-Interface eines Wrappers festgelegt. Da jeder Wrapper über sein eigenes Datenmodell verfügt, welches auf seine jeweilige Datenquelle zugeschnitten wurde, wird hier beschrieben, wie das REST-Interface eines Wrappers in Abhängigkeit von seinem Datenmodell aufgebaut ist. Der Aufbau des Datenmodells wurde bereits in einem vorherigen Kapitel beschrieben.

Da die Datenquelle damit bereits semantisch beschrieben wurde, ist bekannt wie die Daten, die eine Datenquelle zur Verfügung stellt, aussehen. Diese sind in der Form von Klassen und ihren zugehörigen Attributen im RDF-Datenmodell deklariert. Ebenfalls wurde anhand der SPIDER-Ontologie ausgedrückt, welche Semantiken diese Klassen und Attribute im Hinblick auf die Datenquelle besitzen.

Anhand dieser Informationen, lässt sich das Interface für die REST-Zugriffe des Wrappers herleiten. So wird festgelegt, dass für jede im Datenmodell existierende Klasse eine gleichnamige Resource im REST-Interface existieren soll. Für bspw. eine Klasse namens Bauteil würde die URL demnach `http://www.wrapper.org/Bauteil` lauten.

Diese URI agiert als Collection für die entsprechende REST-Resource dieser Klasse. Auf dieser Collection sollen Anfragen gestellt werden können, welche die Klasse im Allgemeinen betreffen, oder Informationen über alle instanziierten Objekte dieser Klasse erhalten. Die genaue Spezifikation für diese Anfragen und die zulässigen HTTP-Methoden auf diese Resource sind in **Tabelle 11** zu entnehmen.

Für die semantische Beschreibung der Datenquellen wurde festgelegt, dass jede Klasse über exakt ein identifizierendes Attribut verfügt (spider:uniqueId). Demnach besitzt jedes instanziierte Objekt vom Typen einer Klasse einen Wert, über den dieses Objekt in der Datenquelle eindeutig identifizierbar ist.

Unter Verwendung dieser ID wird die nächste Stufe des REST-Interfaces der Klasse gebildet. So lassen sich Anfragen zu bestimmten Objekten im System stellen, in dem die ID des Objektes als REST-Resource an die Collection der Klasse angefügt wird. Als Beispiel würde für

Methode	Beschreibung
GET	Liefert eine Liste mit Informationen über alle in der Datenquelle vorhandenen Objekte des Typen dieser Resource. Als Antwort wird ein JSON-LD Dokument gesendet, welches die gelisteten Objekte enthält. Diese sind durch ihre IRI gekennzeichnet. Diese Methode ist nur vorhanden, wenn die semantische Beschreibung dieser Klasse Lese-Zugriffe gestattet (spider:read).
PUT	Diese Methode erstellt ein neues Objekt vom Typen dieser Klasse innerhalb der Datenquelle. Im HTTP-Body der Anfrage muss sich ein valides JSON-LD Dokument befinden, welches unter der Verwendung der Semantik der Klasse, eine instanziiertes Objekt, mit entsprechend ausgefüllten Werten enthält. Diese Methode ist nur vorhanden, wenn die semantische Beschreibung dieser Klasse es gestattet neue Objekte zu erstellen (spider:create).

Tabelle 11: Spezifikation der REST-API für Collections einer entsprechenden Klasse im RDF-Schema eines Wrappers.

ein Objekt vom Typ Bauteil, bei welchem die Teilenummer als ID gewählt wurde, die folgende URL erhalten: `www.wrapper.org/Bauteil/P10128-101`

Diese Resource repräsentiert exakt dieses eine Objekt in der Datenquelle. Mit Anfragen an diese Resource kann man Informationen über dieses Objekt erhalten, Werte dieses Objektes verändern oder dieses Objekt aus der Datenquelle löschen. Die genaue Spezifikation der REST-API für diese Resourcen ist in **Tabelle 12** zu sehen.

Methode	Beschreibung
GET	Liefert eine JSON-LD Repräsentation des Objektes aus der Datenquelle zurück. Dieses Dokument enthält eine Auflistung der Attribute des Objektes, sowie ihrer entsprechenden Werte. Ist in der semantischen Beschreibung der Klasse ein Attribut explizit als nicht lesbar gekennzeichnet, dann wird es in dieser Auflistung nicht aufgeführt (spider:get). Diese Methode ist nur vorhanden, wenn die semantische Beschreibung dieser Klasse Lese-Zugriffe gestattet (spider:read).
POST	Diese Methode erstellt ein neues Objekt vom Typen dieser Klasse innerhalb der Datenquelle. Diese Methode erlaubt es, ein vorhandenes Objekt im Quellsystem zu verändern. Dabei wird im HTTP-Body der Nachricht ein JSON-LD Dokument erwartet, in welchem unter der Verwendung der semantischen Definition der Klasse, die zu verändernden Attribute und ihre neuen Werte aufgeführt sind. Zu verändernde Attribute müssen dabei explizit in der semantischen Beschreibung der Klasse als veränderbar gekennzeichnet worden sein (spider:set) Diese Methode ist nur vorhanden, wenn die semantische Beschreibung dieser Klasse es gestattet vorhandene Objekte zu verändern (spider:update).
DELETE	Führt eine Löschung eines vorhandenes Objekt innerhalb der Datenquelle aus. Diese Methode ist nur vorhanden, wenn die semantische Beschreibung dieser Klasse es gestattet vorhandene Objekte zu löschen (spider:delete).

Tabelle 12: Spezifikation der REST-API für Objekte einer entsprechenden Klasse im RDF-Schema eines Wrappers.

3.8.2 Design und Komponenten

In diesem Abschnitt sollen Überlegungen über den konkreten Aufbau und die benötigten Komponenten für die Realisierung der Wrapper stattfinden.

Ein Wrapper muss in der Lage sein REST-Anfragen zu erhalten und zu verarbeiten. Zusätzlich muss er in der Lage sein, die in den Anfragen enthaltenen JSON-LD Dokumente zu lesen und auch selbst JSON-LD Dokumente zu Erzeugen. Ebenso ist ein Verständnis der SPIDER-Ontologie und der jeweilige semantische Beschreibung seiner Datenquelle erforderlich. Dies ist Funktionalität die für alle Wrapper identisch ist, unabhängig von ihren unterliegenden Datenquellen. Aus diesem Grund ist bei diesen Komponenten das Potenzial für Software-Reuse am höchsten.

Die eigentliche Varianz der Wrapper geht von ihren Datenquellen aus. Jede Datenquelle besitzt ihre eigenen Möglichkeiten diese anzusprechen. Diese proprietären Logiken müssen vom Wrapper implementiert werden. Zusätzlich muss eine Übersetzung der Anfragen an die Wrapper in eine entsprechende Anfrage an die Datenquelle erfolgen, und eine Transformation der erhaltenen Resultate der Datenquelle, in das Datenformat von SPIDER.

Demnach werden für die Wrapper folgende Komponenten identifiziert:

- **JSON-LD Prozessor**

  Diese Komponente ist dafür zuständig, JSON-LD Dokumente zu verarbeiten und Informationen aus diesen auszulesen. Ebenso soll diese Komponente JSON-LD Dokumente erzeugen und verändern können. Neber dem syntaktischen Verständnis von JSON-LD Dokumenten, soll diese Komponente auch für das semantische Verständnis der im SPIDER eingesetzten Ontologien zuständig sein. Dies umfasst die SPIDER-Ontologie, sowie vor allem das semantische Verständnis der Beschreibungen der Datenquellen.

  Diese Komponente wird für die Verarbeitung der Anfragen benötigt, die an den Wrapper gestellt werden.

- **REST Server**

  Diese Komponente ist für die Bereitstellung der REST-Schnittstelle des Wrappers zuständig. Für diese Aufgabe, muss diese Komponente einen HTTP-Server zur Verfügung stellen, welcher in der Lage ist HTTP-Nachrichten zu empfangen, sowie HTTP-Antworten zu versenden. Dabei wird nur benötigt, dass nur die für REST relevanten HTTP-Methoden verarbeitet werden müssen.

  Der REST-Server muss hierbei die Spezifikationen der Schnittstelle erfüllen, wie sie in Kapitel 3.8.1 festgelegt wurden.

- **Datenquellen Anbindung**

  Diese Komponente ist für die eigentliche Anbindung der Datenquelle zuständig. Dafür werden die entsprechenden proprietären Logiken für die Kommunikation mit der Datenquelle verwendet. Zusätzlich ist diese Komponente dafür zuständig die lokalen Datenformate der Datenquelle in das globale Datenformat von SPIDER zu übertragen, sowie auch der umgekehrte Fall.

Das UML-Aktivitäten Diagramm in **Abb. 12** zeigt einen möglichen Ablauf der Verarbeitung eingehender Anfragen. Dabei wurden die Komponenten des REST-Servers und der JSON-LD Verarbeitung in einem *Server*-Layer zusammengefasst. Diese Ebene bildet auch die erste Station für die Verarbeitung eingehender Anfragen. Die Komponente der Datenquellen Anbindung wurde ihren Aufgabengebieten nach weiter aufgespalten. Demnach findet eine Abtrennung zwischen der Transformation der Datenmodelle und der konkreten Anbindung an die Datenquelle statt. Im dadurch entstanden *Data Model*-Layer sollen die weitergereichten Anfragen zunächst in ein lokales Datenformat umgewandelt werden, um dieses dann zur Verarbeitung an die Datenquelle weiterzureichen. Von der Datenquelle erhaltene Resultate werden in dieser Ebene dann ebenfalls wieder in das globale Datenformat von SPIDER umgewandelt. Im *Handler*-Layer letztlich befindet sich die eigentliche Logik zum Ansprechen der Datenquelle.

Diese Trennung ermöglicht es, dass in der Handler-Schicht sich nur um die konkrete Anbindung zur Datenquelle gekümmert muss, ohne dass dabei Einzelheiten der Transformation zwischen den Datenmodellen benötigt werden. Ebenso kann der Data Model Layer sich nur auf

diese Transformationen konzentrieren, ohne über konkrete Logiken der Datenquelle verfügen zu müssen.

Da bereits Schichten identifiziert werden konnten, lässt sich für die generische Architektur des Wrappers eine *Layered Architecture* einsetzen. Diese hat den Vorteil, dass eine klare Isolation zwischen den Layern und ihrer Zuständigkeitsbereiche existiert. Dadurch lassen sich Layer einfach austauschen, sofern sie das entsprechende Interface zur Verfügung stellen. Da der Datenfluss die einzelnen Layer dabei hierarchisch durchläuft, wird das Szenario vermieden, dass ein Layer Informationen eines in der Hierarchie tiefer befindlichen Layers benötigt, welches oft einen Nachteil der *Layered Architecture* bildet. [32]

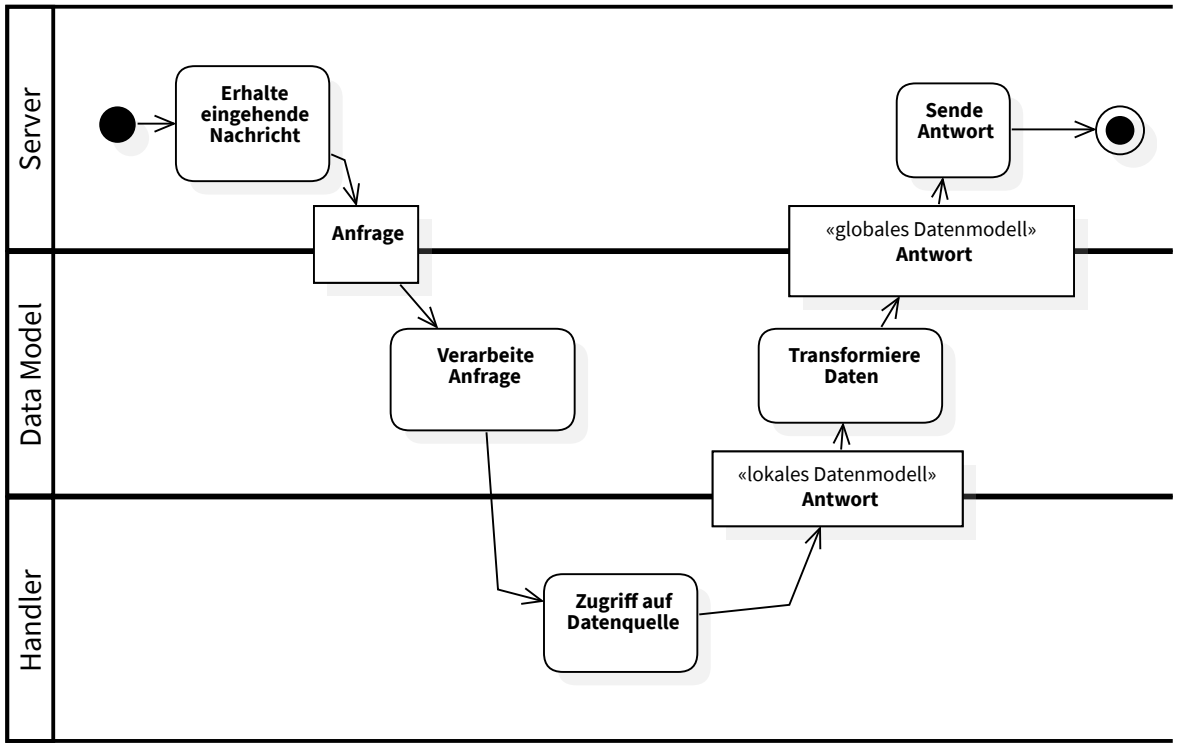

Abbildung 12: UML-Aktivitäten-Diagramm für die Verarbeitung eingehender Nachrichten eines Wrappers

3.8.3 Wrapper-Synthese

Die Unterschiede der Wrapper liegen hauptsächlich bei der Anbindung der Quellsysteme, sowie bei der Bereitstellung der benötigten REST-Interfaces für das lokale Datenmodell.

Im vorherigen Abschnitt konnte eine generische Architektur für einen Wrapper erstellt werden. In diesem Abschnitt soll nun untersucht werden, ob sich diese Architektur, unter Verwendung der semantischen Beschreibung der Datenquelle als Modell, ein für diese Datenquelle angepasster Wrapper instantiieren lässt.

Dabei muss zunächst die Varianz, die durch unterschiedliche Datenmodelle erzeugt wird, festgehalten werden. Eine Möglichkeit dies zu tun, ist durch die Verwendung von Variation

Points. Variation Points identifizieren Stellen an denen es zu einer Variation kommen kann. Typischerweise lassen sich Lösungen aufstellen, um einen Variation Point aufzulösen. Dies kann durch eine simple Ersetzung innerhalb eines Modells passieren, durch Parametrisierung eines Parameters oder durch eine Neuerstellung der betroffenen Komponente. [18]

Dafür werden zunächst die einzelnen Variation Point des Wrappers identifiziert und untersucht, ob diese Varianz sich durch Verwendung des semantischen Datenmodells der Datenquelle auflösen lässt.

Variation Point 1: REST Server

Alle zur Laufzeit eingehenden HTTP-Anfragen an den Wrapper werden zunächst vom REST-Server empfangen. Im Header dieser Nachricht ist die Resource sowie die Methode der Anfrage angegeben. Für jede definierte Klasse in der semantischen Beschreibung der Datenquelle, muss der Wrapper eine REST-Resource zur Verfügung stellen. Der Name dieser Resource entspricht dem Label der definierten Klasse. Demnach ist die Varianz des REST Servers abhängig von den im Datenmodell definierten Klassen. Diese Varianz lässt sich im in **Abb. 13** abgebildeten Aktivitätsdiagramm festhalten. In diesem Diagramm wurde die Varianz mit einem *«Varianz»*-Block festgehalten. Dies entspricht dabei keiner gängigen UML-Notation, und dient nur zu Zwecken der Veranschaulichung innerhalb des Diagramms.

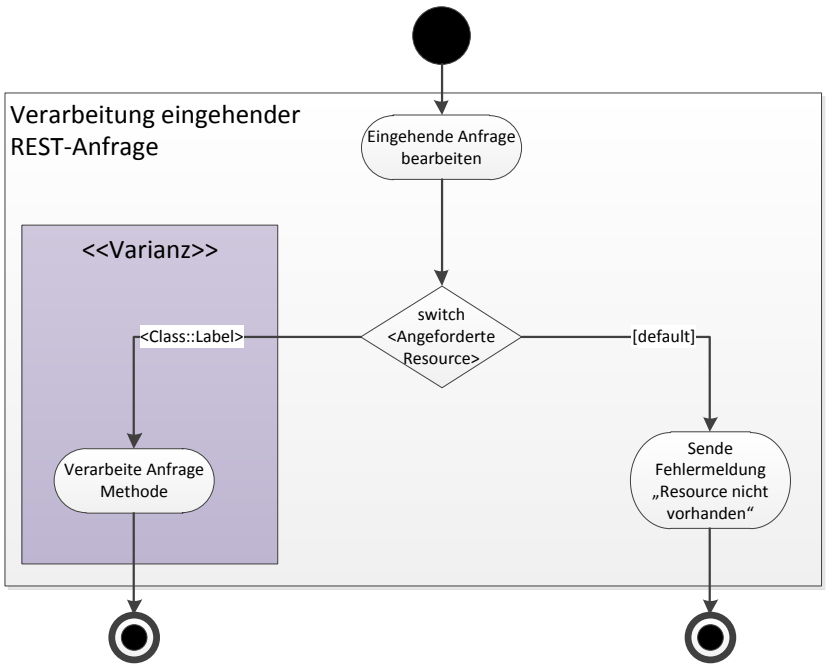

Abbildung 13: Aktivitätsdiagramm für die Bearbeitung eingehender Anfragen des REST-Servers mit eingezeichneter Varianz.

In **Tabelle 13** ist ein Entscheidungsdiagramm abgebildet, mit welchem sich, durch Verwendung des Datenmodells einer Datenquelle, ein Aktivitätsdiagramm für den REST-Server

des Wrappers instantiieren lässt.

	Frage		Diagramm	Effekt
1	Klasse definiert in Datenmodell der Datenquelle?	Y	Aktivitätsdiagramm	Füge neuen Fall in der Fallunterscheidung der Resource hinzu.
		N	Aktivitätsdiagramm	-

Tabelle 13: Entscheidungmodell für die Auflösung der Varianz des Aktivitätsdiagramms aus Abbildung 13

Die nächste Form der Varianz befindet sich bei den erlaubten HTTP-Methoden der Resourcen. Diese sind abhängig, welche erlaubten Zugriffe bei der Klasse mittels der SPIDER-Ontologie definiert wurden. Demnach muss eine Überprüfung stattfinden, ob die angeforderte Methode für diese Resource erlaubt ist. Ein Aktivitätsdiagramm dieses Vorgangs ist in **Abb. 14** zu sehen. Das entsprechende Entscheidungmodell um diese Varianz für eine gegebene Klasse und dem Datenmodell in diesem Diagramm aufzulösen befindet sich in **Tabelle 14**

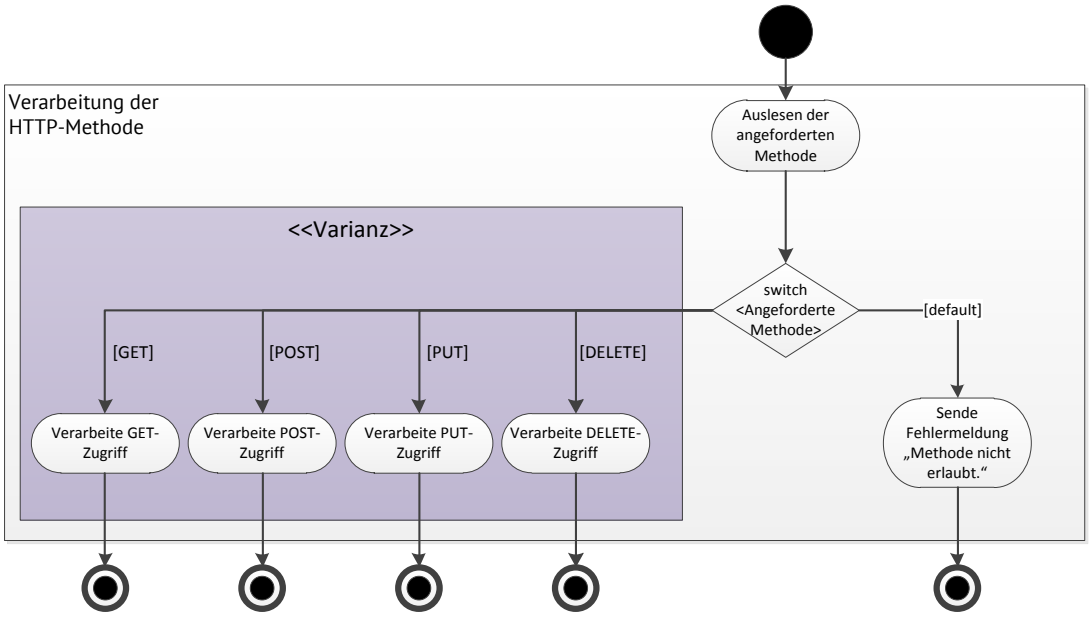

Abbildung 14: Aktivitätsdiagramm für die Verarbeitung der angeforderten HTTP-Methode.

Variation Point 2: Anbindung der Datenquelle

Da jedem Wrapper ein anderes Quellsystem unterliegt, unterschieden sich auch die genauen Logiken für die Datenzugriffe in diese Systeme. Demnach müssen diese Zugriffe für jeden Wrapper, abhängig vom eingesetzten Quellsystem und den Möglichkeiten dieses anzusprechen, eigens implementiert werden. Diese Varianz lässt sich auch nicht mit Hilfe der semantischen Beschreibung der Datenquelle lösen, da diese absichtlich keine Informationen über das

	Frage		**Diagramm**	**Effekt**
1	Wert des Attributs <spider:read> innerhalb der Klassendefinition.	true	Aktivitätsdiagramm	-
		false	Aktivitätsdiagramm	Entferne Fall [GET] aus der Fallunterscheidung.
2	Wert des Attributs <spider:create> innerhalb der Klassendefinition.	true	Aktivitätsdiagramm	-
		false	Aktivitätsdiagramm	Entferne Fall [PUT] aus der Fallunterscheidung.
3	Wert des Attributs <spider:update> innerhalb der Klassendefinition.	true	Aktivitätsdiagramm	-
		false	Aktivitätsdiagramm	Entferne Fall [POST] aus der Fallunterscheidung.
4	Wert des Attributs <spider:delete> innerhalb der Klassendefinition.	true	Aktivitätsdiagramm	-
		false	Aktivitätsdiagramm	Entferne Fall [DELETE] aus der Fallunterscheidung.

Tabelle 14: Entscheidungsmodell für die Auflösung der Varianz innerhalb des Aktivitätsdiagramms von **Abb. 14**

eigentlich unterliegende System besitzt, sondern nur das Datenmodell der Datenquelle mit globalen Konzepten abbildet.

Sind die Zugriffe in die Quellsysteme erfolgt, dann ist der nächste Schritt des Wrappers die Transformation des lokalen Datenformats der Datenquelle in das Format von SPIDER. Es findet also eine Serialisierung dieser Daten in das JSON-LD Format statt. Es wird also ein JSON-LD Dokument erstellt, in dem eine Node vom Typen der definierten Klasse aus der semantischen Beschreibung ist. Dieser Schritt lässt sich anhand von Informationen aus der semantischen Beschreibung der Datenquelle automatisieren, wie das Pseudo-Code Beispiel für den Lese-Zugriff eines Objektes innerhalb der Datenquelle in **Listing 4** zeigt.

Listing 4: Pseudo-Code für einen erzeugten Zugriff in ein Quellsystem anhand der semantischen Beschreibung.

```
GetObject(id)
    objekt = suche_objekt_in_datenquelle(id)
    node = new JsonLdNode()
    node["@id"] = objekt.id
    node["@type"] = objekt.type
    foreach attribute in schema(objekt)
        node[attribute.label] = objekt(attribute.label)
    return node
End
```

4 Implementierung

In diesem Kapitel wird die Implementierung der einzelnen Komponenten von SPIDER beschrieben. Dabei wird aufgelistet welche Programme und Tools bei der Entwicklung verwendet werden.

Es wird beschrieben wie bei der Entwicklung der einzelnen Komponenten vorgegangen wurde und auf spezielle Methodiken oder Gegebenheiten hingewiesen, die bei der Entwicklung angefallen sind.

4.1 Entwicklungsumgebung

Die Entwicklung erfolgte auf einem Microsoft Windows 10 System. Als Entwicklungsumgebung wurde Microsoft Visual Studio 2017 verwendet. Als Testumgebung kamen Microsoft Windows 10 sowie Microsoft Windows Server 2015 zum Einsatz.

4.2 Komponenten

Im Design der Architektur wurden bereits erste Kandidaten für Code-Reuse identifiziert. Um diesen Resue zu ermöglichen sollen diese Komponenten in Form einer Programmbibliothek (oftmals auch Library genannt) implementiert werden. Dabei handelt es sich um Sammlungen von Programmroutinen, auf welche bei der Entwicklung von Programmen zurückgegriffen werden kann. [37] Diese lassen sich durch statisches Linking entweder direkt in das fertige Programm kompilieren, oder durch einen Vorgang namens Dynamic Linking zur Laufzeit des Programms einbinden.

Unter Microsoft Windows sind Programmbibliotheken durch Dynamic-Link Libraries (DLL) vertreten. [29] Eine DLL lässt sich mit Hilfe der Entwicklungsumgebung Visual Studio erzeugen. Dafür muss die Programmiersprache C++ verwendet werden. Für die Erzeugung der DLL muss zusätzlich ein Export Interface angegeben werden, um festzulegen welche Funktionen diese DLL der Außenwelt zur Verfügung stellt. [28]

Andere Anwendungen können zur Laufzeit mittels der C++ Calling Conventions Aufrufe in eine DLL absetzen. Damit ist es möglich eine DLL innerhalb eines Programmes zu verwenden, welches nicht in C++ geschrieben wurde. Voraussetzung dabei ist, dass die Programmiersprache einen Mechanismus zur Verfügung stellt um eine DLL mittels der erwähnten Konventionen zu laden. Dabei muss ein sog. Language Binding für die entsprechende DLL erstellt werden, welche das Interface dieser DLL in die eingesetzte Programmiersprache überträgt. [27]

Eine Möglichkeit für eine automatische Generierung dieser Language Bindings ist die Verwendung des Tools SWIG. [35] Dieses ist in der Lage, unter Verwendung der Header Files und der Interface Beschreibung einer DLL, Language Bindings für eine Vielzahl von Sprachen zu erzeugen.

Im Hinblick auf diesen Mechanismus der Wiederverwendung, wurden der JSON-LD Prozessor und der REST-Server beide in C++ implementiert und innerhalb einer DLL namens *libspider*

zusammengefasst. Libspider findet dabei Verwendung im *Server*-Layer der vorgestellten Wrapper Architektur. Da diese durch Language Bindings von vielen anderen Sprachen eingebunden werden kann, müssen der REST-Server und der JSON-LD Prozessor demnach nur ein einziges mal implementiert werden und stellen damit ein weiteres Core Asset für SPIDER dar.

Die Language Bindings für libspider wurden mit SWIG für die Sprache C# erzeugt. Nach Bedarf lassen sich dabei auch die Bindings für andere Sprachen erstellen.

4.2.1 JSON-LD

Da für JSON-LD keine fertige Implementierung in C++ existiert, wird auf eine eigene Implementierung zurückgegriffen. Diese wurde als statische C++ Library realisiert und lässt sich damit für die weitere Entwicklung von anderen Komponenten wiederverwenden. Für das interne Verarbeiten des JSON Datenformats wird die Open-Source Library JsonCpp verwendet, welche frei unter der MIT License verfügbar ist. [20] Bei der Implementierung der Library wurde die JSON-LD Spezifikation erfüllt. [2]

Das Interface der JSON-LD Hilfsbibliothek stellt drei Klassen zur Verfügung. Die Klasse *Context* dient zur Auflösung und Bearbeitung von JSON-LD *@context* Elementen. Die Klasse *Node* repräsentiert eine Node im Graphen eines JSON-LD Dokumentes. Es lassen sich mit dieser Klasse die Elemente *@id*, *@type* sowie die Attribute der Node auslesen sowie verändern. Die Klasse *Graph* repräsentiert den Graphen eines JSON-LD Dokumentes.

4.2.2 REST Server

Für die Implementierung des REST-Servers wurde auf die Library Boost.Beast aus der Boost C++ Library Sammlung zurückgegriffen. [12] Für den internen Umgang mit JSON-LD Dokumenten innerhalb des Servers wird auf die in SPIDER erstellte JSON-LD Library zurückgegriffen.

Mit der verwendeten Library Beast lies sich ein HTTP-Server implementiert und in der Klasse Server gekapselt. Zum starten des Servers auf einem gegebenen Port stellt das Interface der Klasse die Methode *start()* zur Verfügung. Wurde der Server gestartet, lässt sich mit der Methode *WaitForRequest()* blockierend auf eine eingehende HTTP-Nachricht warten.

Jede eingehende HTTP-Nachricht wird innerhalb eines Objektes vom Typen Request gekapselt. Darin enthalten sind alle Informationen um eine Anfrage abzuarbeiten:

- Die angeforderte Resource
- Die verwendete Methode
- Falls vorhanden, das in einer Anfrage enthaltene JSON-LD Dokument einer Anfrage.

Zusätzlich ist der momentane Zustand der Netzwerk-Verbindung zwischen dem HTTP-Server und dem Client, der die Anfrage gesendet hat, im Request gekapselt. Der erstellte Request wird an der aufrufenden Stelle der Methode *WaitForRequest()* im Code zurückgegeben und kann ab dort zur weiteren Verarbeitung verwendet werden.

Wurde der Request an weiterer Stelle im Wrapper verarbeitet, lässt sich mittels der Methoden *SendSuccess()* und *SendError()* eine HTTP-Antwort versenden. Dabei werden die vom Request gespeicherten internen Verbindungsdaten verwendet.

4.3 Wrapper

In diesem Abschnitt soll eine prototypische Implementierung eines generischen Wrappers beschrieben werden. Dabei wird ebenfalls auf die Schritte eingegangen, wie aus dem entwickelten Wrapper die Stubs und Templates für den Generator erstellt wurden.

4.3.1 Prototypische Implementierung

Das Ziel dieser Implementierung war es, einen möglichst generischen Wrapper zu Erstellen, welcher als Ausgangsbasis für die modellbasierte Generierung von Wrappern verwendet werden könnte. Für die Realisierung wurde dabei die vorgeschlagene Architektur für einen Wrapper aus Kapitel 3.8.2 umgesetzt. Die Funktionalitäten des REST-Servers und des JSON-LD Prozessors wurden mittels libspider als dynamisch-verlinkte Library eingebunden. Dafür wurden die C# Language Bindings verwendet.

Als provisorisches Datenmodell für den Wrapper wurde eine simple Klasse namens *Object* mit einem Attribut vom Typ String gewählt. Anhand diesem Datenmodell wurden die einzelnen Diagramme der variablen Architektur instantiiert, indem die Entscheidungsmodelle angewendet wurden. Eine Anbindung an eine Datenquelle bei der Implementierung des generischen Wrappers wurde nicht vorgenommen.

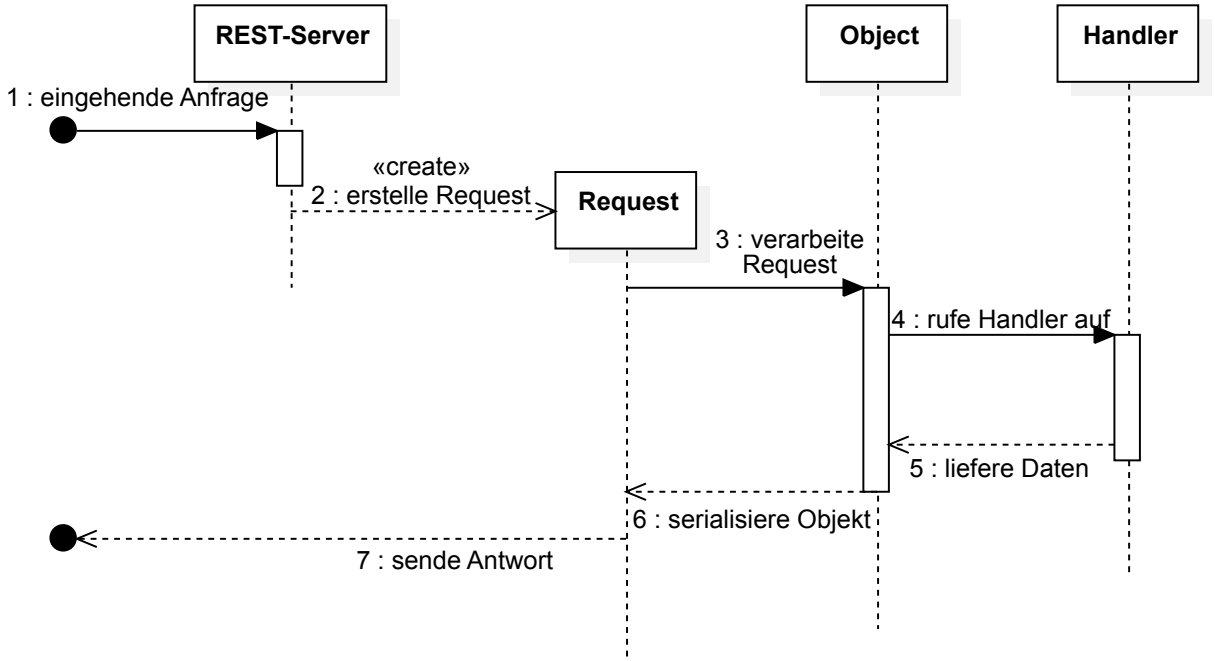

Abbildung 15: UML-Sequenz Diagramm für die Verarbeitung eingehender Nachrichten an den Wrapper.

Realisiert wurde die Implementierung mittels drei Klassen.

- **Server:**

  Diese Klasse kapselt die Funktionalität des REST-Servers von libspider und nimmt eingehende Anfragen entgegen.

Eingehende Anfragen werden als Objekte des Typen *Request* zur Verarbeitung im Server weitergereicht. Dabei findet zunächst eine Überprüfung der angeforderten Resource des Requests statt. Diese wird in einer switch-case Anweisung verarbeitet.

Eingehende Anfragen an die Resource *Object* werden zur weiteren Verarbeitung an eine statische Methode der Klasse *Object* weitergereicht. Anfragen an andere Resourcen werden mit einer Fehlermeldung abgelehnt.

- **Object:**

  Dies ist eine Implementierung des JSON-LD Datenmodells der Klasse Object als eine C# Klasse. Attribute aus der JSON-LD Klassendefinition wurden dabei mit internen Datentypen von C# nachmodelliert.

  Zusätzlich besitzt diese Klasse Methoden zur Serialisierung und Deserialisierung von und zu einem JSON-LD Dokument. Eine statische Methode namens *handleRequest* nimmt ein Objekt vom Typ *Request* und führt dort den nächsten Schritt der Verarbeitung aus. Dort wird, je nach angeforderter Art des Zugriffs (Get, Update, etc.) das weitere Vorgehen entschieden. Zugriffe in die Datenquelle werden dabei delegiert an die Klasse *Handler*.

- **Handler:**

  Diese Klasse kapselt alle Zugriffe in die Datenquelle, die von der Klasse *Object* gestellt werden. Dafür werden statische Methoden zur Verfügung gestellt die für die jeweiligen CRUD-Zugriffe in die Datenquelle zuständig sind.

Ein UML-Sequenz-Diagramm welches diese Abläufe abbildet ist in **Abb. 15** zu sehen.

4.3.2 Ableitung der Templates

Für eine Verwendung des Generators werden Template-Dateien verwendet. Diese Templates sind speziell angepasste Quelltexte, an denen durch annotierte Stellen festgelegt ist, was für Ersetzungen der Generator an dieser Stelle vornehmen soll.

Für die Erstellung der Template-Dateien für die Sprache C# wurden die erstellen Quelltexte des prototypischen Wrappers verwendet. Dort wurden zunächst alle Vorkommnisse des Klassen-Identifiers *Object* durch die Annotation @@*OBJECT*@@ ersetzt.

Anschließend wurden im Quelltext die in Kapitel 3.8.3 aufgestellten Varianzpunkte identifiziert.

Dies war zum Einen die switch-case Anweisung der Anfrageverarbeitung der Server Klasse, welche in **Listing 5** zu sehen ist. Diese entspricht dabei dem UML-Aktivitätsdiagramm aus **Abb. 13**.

Listing 5: Identifizierter Varianzpunkt innerhalb des Quelltextes der Klasse Server.

```
1 private void HandleRequest(Request request)
2 {
3     switch(request.getResource())
```

```
4       {
5           case "Object":
6               Object.HandleRequest(request);
7               break;
8           default:
9               request.SendError("Requested resource not in Schema!");
10              break;
11      };
12  }
```

Diese Stelle des Codes legt die gültigen Resourcen für den REST-Server fest. Demnach muss für jede Resource in dieser switch-case Anweisung ein Eintrag angelegt werden.

Um dies für den Generator sichtlich zu machen, wurde der case-Block entfernt und mit der Annotation *@@RESOURCE_CASES@@* ersetzt. Der entfernte Block wurde dabei als Pattern für den Einsatz an dieser Stelle im Generator vermerkt.

Eine ähnliche Ersetzung wurde bei der switch-case Anweisung in der Klasse **Object** vorgenommen, welche eine Überprüfung der Resource des Requests vornahm. Diese Stelle entspricht dem Aktivitätsdiagramm aus **Abb. 14**.

Zusätzlich wurde aus der Klasse *Object* ein Template für die Generierung einer Klassendefinition erzeugt. Diese Template-Datei wurde dieser Arbeit in **Anhang A** beigefügt.

4.4 Generator

Vor der Implementierung des Generators wurde zunächst eine Recherche über bereits existierende Code-Generator Frameworks getätigt. Einer der bekanntesten Vertreter dieser Art ist die Xtext Grammar Language der Eclipse Foundation. Diese ist ein Plugin für die Entwicklungsumgebung Eclipse und kann anhand einer vorher angefertigten Grammatik Text-Bausteine generieren.

Da man dadurch zunächst eine Grammatik definieren müsste, welche das JSON-LD Format parsen kann, sowie das Verständnis der semantischen Informationen der SPIDER-Ontologie implementieren müsste, wurde sich gegen diese Lösung entschieden. Stattdessen fiel die Entscheidung auf eine Implementierung eines eigenen Generators in C++, um auf die bereits vorhandene Funktionalität von libspider zurückzugreifen. Im Folgenden ist die Implementierung des Generators beschrieben.

Der Generator selbst ist ein ausführbares Kommandozeilen-Programm, welches die semantische Beschreibung einer Datenquelle in Form einer oder mehrerer JSON-LD Dokumente als Eingabedateien erhält. Über einen Kommandozeilen-Parameter lässt sich zudem die gewünschte Zielsprache der erzeugten Quelltexte festlegen.

Um die benötigten Stubs für den Wrapper zu erzeugen, greift der Generator auf mehrere Template-Dateien zurück. Diese Template Dateien sind speziell angepasste Quelltexte, an denen durch Annotationen gekennzeichnet ist, welche Teile der Generator anpassen muss. Um eine Erweiterbarkeit des Generators zu gewährleisten, ist es möglich neue Programmiersprachen hinzuzufügen, in dem die jeweiligen Template Dateien für die Stubs dieser Sprache im entsprechenden Ordner des Generators abgelegt werden. In dieser Arbeit wurden nur Templates für die Programmiersprache C# angefertigt.

Für die Erzeugung des Wrappers werden vom Generator drei Template Dateien benötigt:

- **Server.Template:**

  Dieses Template beinhaltet den Code für die Ausführung des REST-Servers und wird für jeden Wrapper genau einmal erzeugt. Eingehende REST-Anfragen werden von der aus diesem Template erzeugten Klasse abgehandelt.

  In **Listing 6** ist ein Ausschnitt aus der Server Template-Datei für die Sprache C# zu sehen. Die dort enthaltene switch-case Anweisung entspricht den REST-Resourcen die dieser Server zur Verfügung stellt. Für jede definierte Klasse im Schema wird demnach ein neuer Eintrag an die mit der Annotation *HANDLER_CASES* gekennzeichnete Stelle erzeugt. Dies entspricht dem Entscheidungsmodell aus **Tabelle 13** von Kapitel 3.8.3.

  Ein Beispiel für einen generierten Eintrag ist als Kommentar in diesem Ausschnitt zu sehen. Die vollständige Template Datei wurde dieser Arbeit in **Anhang C** angefügt.

Listing 6: Ausschnitt aus der Template-Datei Server.Template.cs für C#

```
private void HandleRequest(Request request)
{
    switch(request.getObject())
    {
        @@HANDLER_CASES@@
//      case "Part":
//          Object.Part.HandleRequest(request);
//          break;
        default:
            request.SendError("Object not in Schema!");
            break;
    };
}
```

- **Object.Template:**

  Für jede im Schema definierte Klasse wird vom Generator eine Klasse im Quelltext des Wrappers erzeugt. Die Template Datei Object.Template wird hierfür verwendet. In dieser befindet sich das Grundgerüst für eine Klasse innerhalb der gewünschten Programmiersprache. Dabei wird zunächst der Name der Klasse bestimmt durch das <rdf:label> Property aus der Schemadefinition der Klasse. Anschließend wird durch alle Nodes vom Typ <rdf:Property> im Schema iteriert, welche dieser Klasse zugeordnet sind.

  Für jede dieser Properties wird ein Attribut in der Klasse erzeugt. Der Name wird dabei ebenfalls dem Label der Property entnommen. Im Falle von einer stark-typisierten Sprache, wird noch ein Typ für das erzeugte Attribut angegeben. Dieser wird zunächst aus dem <schema:rangeIncludes> Attribut des jeweiligen Properties ausgelesen. Anschließend findet vom Generator ein Mapping statt, um den Schema-Typen in einen Standard-Typen der Programmiersprache umzuwandeln. Dieses Mapping muss für jede Sprache, die der Generator unterstützen soll, angepasst werden. Ein solches Mapping für C# ist in **Tabelle 15** zu sehen.

Schema	C#
rdf:text	String
xsd:integer	int
xsd:float	float
xsd:double	double
xmls:date	DateTime

Tabelle 15: Mapping zwischen Schema-Typen und Standardtypen von C#

Weiterhin besitzen die generierten Klassen Methoden zur Serialisierung und Deserialisierung in das JSON-LD Datenformat. Ein Ausschnitt für die Serialisierungsfunktion aus der Template-Datei für C# ist in **Listing 7** zu sehen. Dabei wurde das Verfahren aus **Listing 4** in Kapitel 3.8.3 implementiert.

Listing 7: Ausschnitt der Serialisierungsfunktion aus Object.Template.cs

```
class @@OBJECT@@ {
 ...
    private Graph serialize()
    {
        Node node = new Node();

        node.setID(this.CreateNodeID(info));
        node.setType(Program_Part.TypeID);

        @@SERIALIZE_SINGLE_PROPERTIES@@

        Graph graph = new Graph(new Context());
        graph.AddNode(node);

        return graph;
    }
...
}
```

Zusätzlich enthält die generierte Klasse eine statische Methode, an welche die vom REST-Server empfangenen Anfragen weitergeleitet werden, um diese weiter zu verarbeiten. Dort findet die Überprüfung und Ausführung der jeweiligen HTTP-Methode statt. Dieses Verhalten wird ebenfalls durch die im Schema beschriebenen Semantiken der Klasse und ihrer Attribute festgelegt.

In **Listing 8** ist ein Ausschnitt dieser Funktion aus der Template-Datei für C# zu sehen. In diesem Template wird, je nach Definition der Klasse im Schema, vom Generator gewählt, ob die Anfrage weiterverarbeitet wird oder ob stattdessen eine Fehlermeldung gesendet wird. Dieses Verhalten entspricht dem Entscheidungsmodell von **Tabelle 14** aus Kapitel 3.8.3.

Listing 8: Ausschnitt der Anfragebearbeitung der Objekte aus Object.Template.cs

```
class @@OBJECT@@ {
...
    public static void HandleRequest(Request request)
    {
        switch (request.getMethod())
        {
        ...
        case "POST": // Edit object with given ID and params
            @@UPDATE_ALLOWED@@
                Handler.@@OBJECT@@.Update(request.getID(), @@OBJECT@@.
                    fromRequest(request));
                request.SendSuccess();
                break;
            @@ELSE@@
                request.SendError("Update on Type @@OBJECT@@ not allowed!");
                break;
            @@UPDATE_ALLOWED@@
```

```
17          ...
18              default:
19                  request.SendError("Method not supported.");
20                  break;
21          }
22      }
23  }
```

- **Handler.Template:**

  Für jede im Schema definierte Klasse wird zusätzlich ein Handler erzeugt. Dieser Handler ist für die eigentlichen Zugriffe in die Datenquelle des Wrappers zuständig und muss durch den Betreiber angepasst werden.

  In diesem Stub befinden sich fünf statische Methoden, List,Get,Update,Create und Delete, in welche jeweils die entsprechende Logik der Datenquelle eingebaut werden muss. Die Interfaces der Methoden sowie deren Eingangs- und Ausgangsflüsse sind dabei bereits vorgegeben.

  Ein Ausschnitt dieser Stubs aus der Template-Datei für C# ist in **Listing 9** zu sehen. Die Stellen an welche der Code eingefügt werden soll sind durch Kommentare gekennzeichnet.

  Im Falle der Methode *Get* muss Code eingefügt werden, welcher ein Objekt mit der gegebenen ID aus der Datenquelle ausliest, und die Daten aus dem lokalen Datenformat der Quelle in die vorgegebene Instanz des Objektes *obj* überträgt. Dabei hat der Programmierer Zugriff auf die generierten Member-Attribute der Klasse. Die Serialisierung des Objektes sowie das Senden der Antwort wird anschließend von der Infrastruktur des Wrappers vorgenommen, sobald die Methode verlassen wird.

  Im Falle der Methode *Update* wird vom Programmierer verlangt, dass dieser Code zur Verfügung stellt, welcher das Objekt in der Datenquelle mit der gegebenen ID mit den Daten des übergebenen Objektes *obj* überführt. An dieser Stelle wurde die Deserialisierung der Anfrage bereits vorgenommen, so dass alle benötigten Informationen bereits im übergebenen Objekt sind. Sollte es bei diesem Vorgang zur Laufzeit zu Problemen kommen, kann der Programmierer dies durch das Werfen einer Exception bekannt machen.Der unterbrochene Kontrollfluss wird

Listing 9: Ausschnitt der Template-Datei Handler.Template.cs

```
1  namespace @@PROJECT@@.Handler
2  {
3      class @@OBJECT@@
4      {
5          ...
6          // Return object of given ID
7          public static Object.@@OBJECT@@ Get(String ID)
8          {
```

```
9               Object.@@OBJECT@@ obj = new Object.@@OBJECT@@();
10
11              // Insert code here
12                          throw new NotImplementedException("Not implemented!");
13
14              return obj;
15          }
16
17          // Update object with ID the data from the given object
18          public static void Update(String ID, Object.@@OBJECT@@ obj)
19          {
20              // Insert code here
21                          throw new NotImplementedException("Not implemented!");
22
23                  }
24          ...
25      }
26  }
```

4.5 SPIDER Core

Für die Implementierung des SPIDER-Core wurde auf das Customizing einer Aras Innovator zurückgegriffen. Dies geschah aus dem Hintergrund, dass Aras Innovator als PLM Plattform bereits den Mittelpunkt der PLM Abläufe eines Unternehmens bildet und damit auch im Hinblick auf zukünftige Entwicklungen von SPIDER eine solide Grundlage für die Produkt- und Prozessintegration bildet. Als denkbare Alternativen kämen auch Graphdatenbanken als Backbone in Frage, da das in SPIDER verwendete JSON-LD Datenformat ebenfalls auf Graphen basiert.

Dafür wurde eine Aras Innovator Instanz auf einem Windows Server 2015 installiert. Als Datenbank-Backbone für diese Installation wurde ein Microsoft MSSQL-Server 2016 installiert, sowie ein Microsoft Internet Information Services 7.5 als Webserver eingesetzt.

Durch den Einsatz von Aras Innovator ist es möglich eine Verwaltung der Quellsysteme direkt über die Benutzeroberflächen des mitgelieferten Web-Clients vorzunehmen. [13] Dafür wurde zunächst über die Customizing Möglichkeiten von Aras Innovator ein benutzerdefinierter ItemType für Datenquellen definiert. Dieser ItemType wurde um entsprechende Attribute erweitert, wie den Namen sowie die Netzwerkadresse des Wrappers der Datenquelle. Anschließend wurde eine Eingabemaske für diesen ItemType erstellt, welche in **Abb. 16** zu sehen ist.

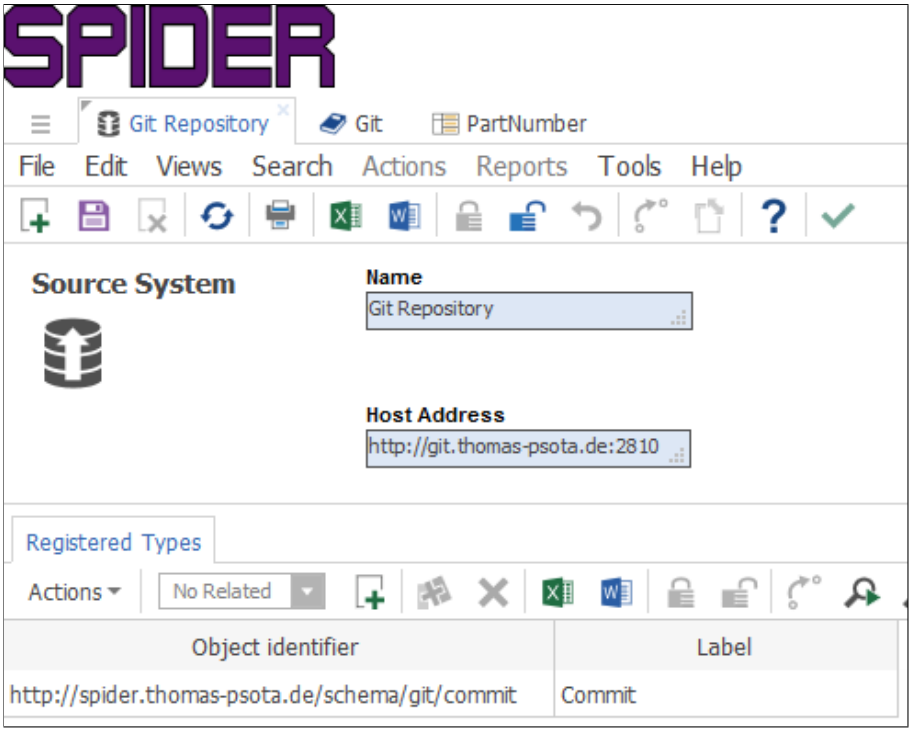

Abbildung 16: Eingabemaske für Quellsysteme in der angepassten Aras Innovator Instanz

In dieser Eingabemaske sind die Eingabefelder für den Namen und die Netzwerkadresse. Im unten befindlichen Reiter ist eine Liste, in welche man die Klassen, die diese Datenquelle zur Verfügung stellt eintragen kann. Dabei kann man aus den im SPIDER-Core definierten Klassen auswählen.

Um eine Beschreibung der Datenmodelle der Quellsysteme über die Oberfläche zu ermög-

lichen, wurde die Aras Innovator Instanz um Möglichkeiten zur Modellierung von Klassen und Properties, wie sie im Datenmodell von SPIDER eingesetzt werden, erweitert. Dafür wurden weitere benutzerdefinierte ItemTypes namens Schema, Class, Property und BaseType angelegt.

Mit dem ItemType Schema ist es möglich Sammlungen von Ontologien anzulegen. Dabei kann bei einem Eintrag unterschieden werden, ob es sich um ein selbst-definiertes oder um ein fremdes Vokabular handelt. Für fremde Vokabulare wird die URL des Schema Dokumentes eingetragen. Für selbst-definierte Vokabulare wird von Aras Innovator eine URL sowie das Schema Dokument im JSON-LD Format selbst erzeugt. Ein Screenshot der Eingabemaske für diesen ItemType ist in **Abb. 17** zu sehen. Darauf abgebildet sind die Eingabefelder der einzelnen Werte, sowie eine Auflistung der zu diesem Schema gehörenden Schema Elemente.

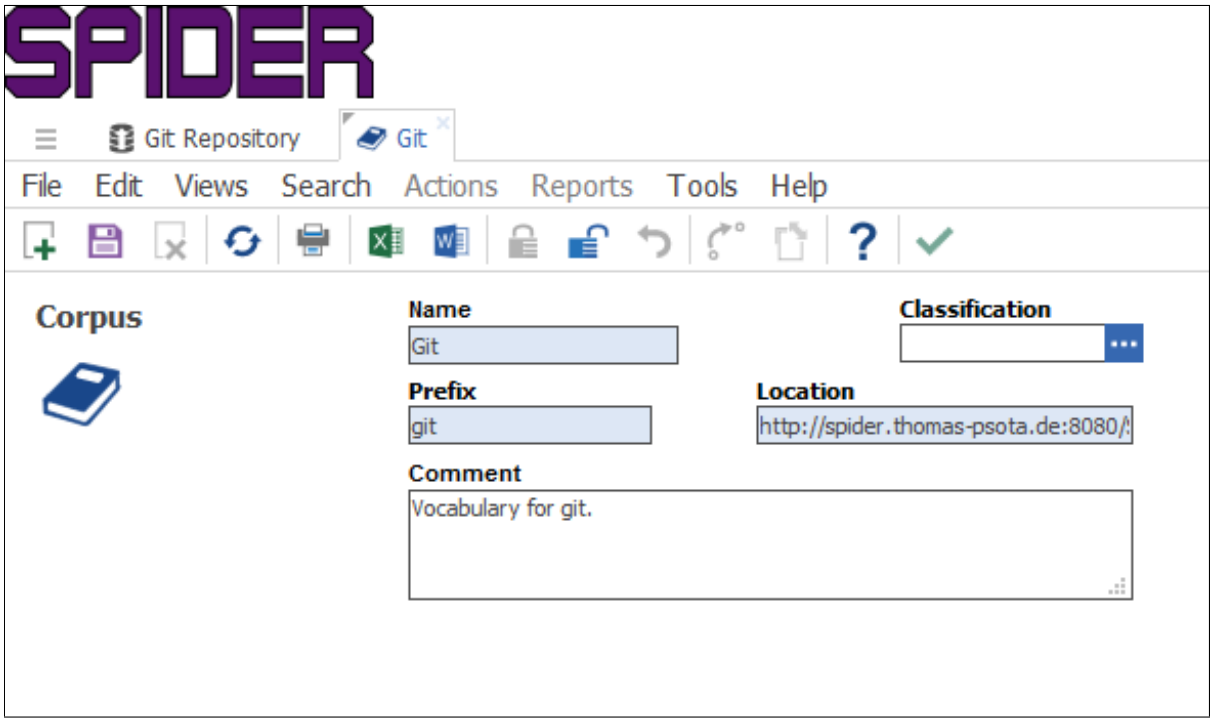

Abbildung 17: Screenshot der Eingabemaske des ItemType Schema in der angepassten Aras Innovator Instanz

Die Modellierung der einzelnen Klassen für das Schema einer Datenquelle wird über den ItemType Class vorgenommen. Die Eingabemaske für diesen ItemType ist in dem Screenshot von **Abb. 18** zu sehen. Dort lässt sich zunächst ein Name und ein Kommentar für die Klasse festlegen. Über das Kontrollelement Schema lässt sich dieser Klasse ein in der Instanz angelegtes Schema zuweisen. Über Checkboxen lässt sich einstellen, welche Art von Operationen auf dieser Klasse erlaubt sind. Dies entspricht den Eigenschaften für Klassen aus der SPIDER-Ontologie. Über den unten befindlichen Reiter Properties lassen sich dieser Klasse ihre Attribute zuweisen. Bei diesen Properties lassen sich ebenfalls über Checkboxen die entsprechenden Zugriffe einstellen.

Die Properties einer Klasse werden über den gleichnamigen ItemType in der Aras Innovator

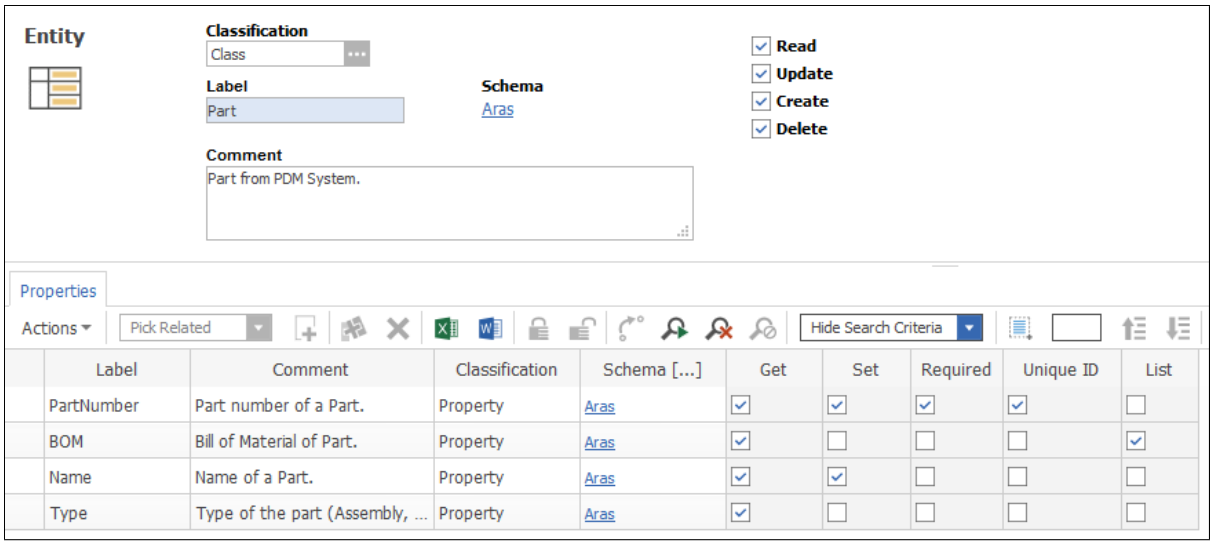

Abbildung 18: Screenshot der benutzerdefinierten Eingabemaske für Klassen in der Aras Innovator Instanz.

Instanz verwaltet. Die Eingabemaske für ein Property ist in **Abb. 19** zu sehen. Für diese wird ebenfalls ein Label und ein Kommentarfeld angezeigt. Zusätlich lässt sich eine Zuweisung zu einem vorher definierten Schema vornehmen. Im unten befindlichen Reiter Type Range lässt sich der Datentyp für dieses Property auswählen. Dies kann eine der in der Instanz definierten Klassen oder ein Basis Typ aus einem anderen Vokabular sein.

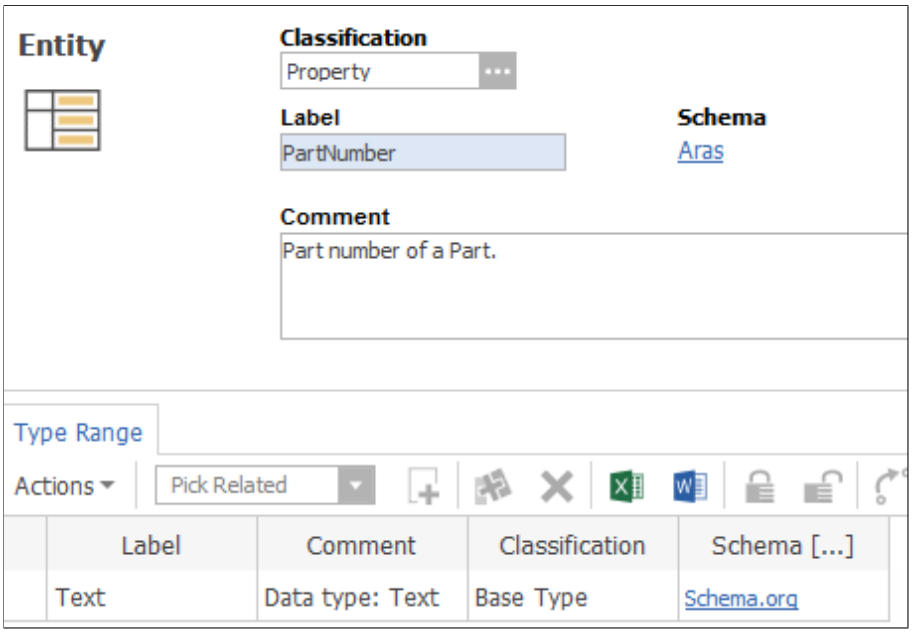

Abbildung 19: Screenshot der Eingabemaske für Properties in der angepassten Aras Innvator Instanz

Um die Verwendung von Basis Typen aus anderen Vokabularen zu ermöglichen, muss für diese ein entsprechender Eintrag über den ItemType BaseType angelegt werden. Dabei wird, wie bei Class und Property, über die Eingabemaske das Label, ein Kommentar, sowie die Zuweisung zum entsprechenden Standard Vokabular vorgenommen.

Um das für den SPIDER-Core spezifizierte REST-Interface einzuhalten wurde mit dem Generator ein Wrapper erzeugt, welcher seine Daten aus der SPIDER Aras Innovator ausliest.

Dieser Wrapper implementiert die REST-Interfaces für die Resourcen SourceSystem und Schema. Anfragen an SourceSystem liefern eine Liste aller momentan an SPIDER angebundenen Datenquellen zurück, indem die Konfiguration der Datenquellen aus der Aras Innovator SPIDER Instanz ausgelesen werden.

Die vom Wrapper zur Verfügung gestellte Resource *Schema* erlaubt Anfragen nach den in der Aras Innovator Instanz konfigurierten Klassen und Schema Informationen. Das Schema in Form eines JSON-LD Dokumentes wird dabei zur Anfragezeit generiert, indem die Klassen und Schema Konfiguration aus der Aras Innovator Instanz ausgelesen und transformiert wird. Der Pseudo-Code um diese Transformation durchzuführen ist dabei in **Listing 10** zu sehen.

Listing 10: Pseudo Code für die Erzeugung eines JSON-LD Dokumentes aus der Schema Konfiguration der SPIDER Aras Innovator Instanz

```
graph = new Graph()
graph["@context"] = schema

foreach class in schema
    node = new Node()
    node["@id"] = schema.prefix + class.label
    node["@type"] = "rdfs:class"

    node["spider:read"] = class.allow_read?
    node["spider:update"] = class.allow_update?
    node["spider:delete"] = class.allow_delete?
    node["spider:create"] = class.allow_create?

    graph.add(node)
end

foreach property in schema
    node = new Node()
    node["@id"] = schema.prefix + property.label
    node["@type"] = "rdf:property"

    node["schema:rangeIncludes"] = property.typeRange
    node["schema:domainIncludes"] = property.class

    node["spider:get"] = property.allow_get?
    node["spider:set"] = property.allow_set?
    node["spider:unique_id"] = property.is_unique_id?
    node["spider:required"] = property.is_required?
end
```

5 Evaluierung

In diesem Kapitel soll eine Evaluierung der erarbeiteten Architektur stattfinden. Dafür wird zunächst in einem ersten Test die Anbindung eines PDM Systems an SPIDER erprobt. An der beispielhaften Anbindung eines CASE-Tools wird zusätzlich eine Performance Analyse für die Verarbeitungszeit von Anfragen an SPIDER vorgenommen. Anschließend werden in einem letzten Test Änderungen an einer angeschlossenen Datenquelle simuliert.

5.1 Anbindung eines PDM Systems

In diesem Abschnitt wird die Anbindung eines PDM Systems an SPIDER erprobt. Als zu integrierendes PDM System wird hierbei Aras Innovator verwendet, da dies aus lizenztechnischen Gründen als Einzigstes für Testzwecke zur Verfügung stand. Da keine realen Testdaten zur Verfügung gestellt werden konnten, wurde für diese Testzwecke eine simple Stückliste im PDM System angelegt.

Für eine Anbindung eines neuen Quellsystems an SPIDER werden dabei folgende Schritte ausgeführt:

1. Einen neuen Eintrag für das Quellsystem im SPIDER-Core anlegen.
2. Festlegen, welche Daten das Quellsystem für die Integration zur Verfügung stellen soll.
3. Modellierung des Datenmodells unter Zuhilfenahme der Modellierungstools im SPIDER Core.
4. Generierung der Schema Dateien durch SPIDER-Core.
5. Generierung des Wrappers mit Input der Schema Dateien.
6. Einfügen der Logiken für den Zugriff in die Datenquelle in den Wrapper-Stub.

5.1.1 Anbindung der Datenquelle

Einer der Hauptzweck eines PDM Systems ist die Verwaltung von Produkten sowie ihrer Produktstrukturen. Für diese Aufgabe stellt Aras den ItemType *Part* zur Verfügung. Dieser enthält die Stammdaten des Bauteils, sowie die Stückliste, falls es eine gibt. Für die Integration sollen daher folgende Daten zur Verfügung gestellt werden:

- Die Teilenummer
- Der Name des Bauteils
- Der Typ des Bauteils (Einzelteil, Baugruppe)
- Die Stückliste des Bauteils

Als ersten Schritt muss die Datenquelle zunächst am SPIDER-Core registriert werden. Dafür wird ein neuer *Source System* Eintrag über die Oberfläche von SPIDER-Core erzeugt. Für diesen wird ein Name und die Netzwerkadresse, auf welcher der spätere Wrapper zu erreichen sein wird, eingetragen.

Als nächstes muss das Datenmodell für das neu angelegte Quellsystem konfiguriert werden. Dafür werden die von SPIDER zur Verfügung Modellierungstools in der Web-Oberfläche verwendet. Eine fertige Konfiguration für die oben erwähnten Daten des Bauteils ist auf dem Screenshot in **Abb. 20** zu sehen.

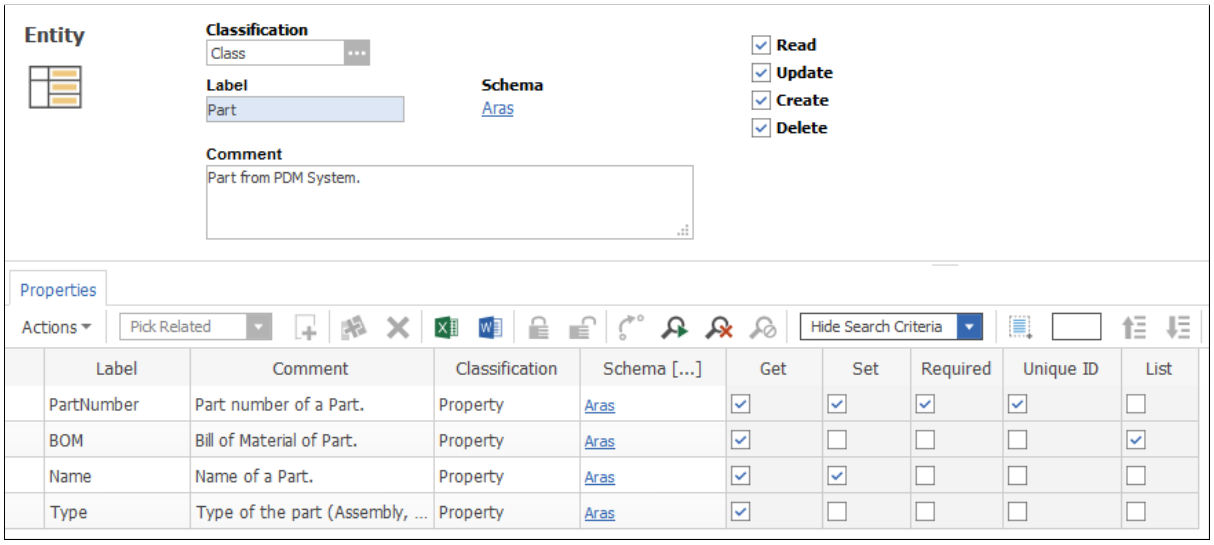

Abbildung 20: Screenshot der Konfiguration der Aras Part Klasse.

Dabei wurde jedem der verwendeten Schema Elemente das Schema **aras** zugewiesen. Die Klasse wurde mit Berechtigungen für alle CRUD-Zugriffe angelegt. Als identifizierendes Attribut wurde die Teilenummer gewählt. Diese wird auch als alleiniges Attribut für eine Erstellung neuer Bauteile benötigt. Da eine Stückliste aus mehreren Bauteilen bestehen kann, wurde diese als Liste deklariert.

Die von SPIDER-Core erzeugte URL für das Schema dieser Klasse lautet *http://spider.thomas-psota.de/schema/aras/part*. Unter dieser URL lässt sich das generierte JSON-LD Dokumente mit der Definition für diese Klasse abrufen. Eine Kopie dieses erzeugten Dokumentes wurde dieser Arbeit in **Anhang D** beigefügt.

Mit diesem erzeugten Dokument lässt sich als Input für den Generator verwenden um einen Stub für einen Wrapper zu generieren. Für die Anbindung zu Aras Innovator wird die bei der Installation mitgelieferte IOM.dll verwendet. Diese ist eine C#-Klassenbibliothek welche Funktionen zur Interaktion mit einer Aras Innovator Instanz bereit stellt. [13] Aus diesem Grund wird ein Wrapper für die Sprache C# generiert.

Ein Aufruf des Generators mit den Parametern für C# und dem erzeugten Schema als Input generiert einen Stub für den Wrapper wie er in Kapitel 4.4 beschrieben wurde. In diesen Stub werden die benötigten Aufrufe geschrieben, um die auf dem Bauteil definierten Operationen in Aras Innovator umzusetzen. Eine Implementierung des *Get* Aufrufs ist in **Listing 11** zu sehen.

Listing 11: Implementierung des Wrapper-Codes für einen Get-Zugriff auf ein Bauteil.

```
// Return object of given ID
public static Object.Part Get(String ID)
{
    Object.Part obj = new Object.Part();

    var partQry = Global.aras.inn.newItem("Part", "get");
    partQry.setProperty("item_number", ID);
    var arasPart = partQry.apply();

    part.PartNumber = arasPart.getProperty("item_number");
    part.Name = arasPart.getProperty("name");
    part.Type = arasPart.getProperty("classification");

    arasPart.fetchRelationships("Part BOM");
    var part_bom = arasPart.getRelationships("Part BOM");
    for (int i = 0; i < part_bom.getItemCount(); ++i)
    {
            var component = part_bom.getItemByIndex(i).getRelatedItem();
            var componentPartNo = component.getProperty("item_number");

            part.BOM.Add(new Object.Part(componentPartNo));
    }

    return obj;
}
```

5.1.2 Inbetriebnahme der Anbindung

Der erstellte Wrapper wird auf der Netzwerkadresse *http://aras.thomas-psota.de* auf dem Port 6112 unter Betrieb genommen und verbindet sich direkt mit der ihm unterliegenden Aras Innovator Instanz. Für den Wrapper wurde ein eigener Benutzer-Account im PDM System angelegt, unter welchem seine Operationen ausgeführt werden.

5.1.3 Testen von Get-Zugriffen

Zur Erprobung wird eine Anfrage nach einem im System liegenden Bauteil mit der Teilenummer P1028 unternommen. Dafür wird eine HTTP-Anfrage vom Typen GET auf die Resource *http://aras.thomas-psota.de/Part/P1028* abgesendet. Die vom Wrapper gesendete Antwort ist in **Listing 12** zu sehen.

Listing 12: Get-Zugriff auf ein Bauteil.

```
{
    "@context" : {
        "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/"
    },
    "@id" : "http://aras.thomas-psota.de:6112/Part/P1028",
    "@type" : "aras:Part",
    "aras:BOM" :
```

```
    [
        {
            "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-101",
            "@type" : "aras:Part"
        },
        {
            "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-102",
            "@type" : "aras:Part"
        }
    ],
    "aras:Name" : "Rotor Frame",
    "aras:PartNumber" : "P1028",
    "aras:Type" : "Assembly"
}
```

In dieser Antwort ist eine Instantiierung der definierten Klasse *aras:Part* zu erkennen.

Das *@id* Attribut enthält die IRI dieses Bauteils im semantischen Netzwerk von SPIDER. Diese IRI ist sowohl eindeutig innerhalb ihrer Datenquelle als auch im globalen Graphen von SPIDER. Im ersten Fall liegt dies an der Semantik der verwendeten Teilenummer bei der von sich aus garantiert ist, dass es kein anderes Bauteil mit einer identischen Teilenummer geben kann. Im zweiten Fall bildet die Kombination aus URL der Datenquelle, Resource, sowie der gewählten ID eine eindeutige Schlüsselkombination.

Die restlichen Attribute enthalten ihre entsprechenden Werte, welche aus der Datenquelle stammen. In der Stückliste bei *aras:BOM* befindet sich eine Liste von weiteren *aras:Part* Einträgen mit ihrer IRI. Diese IRI kann verwendet werden, um weitere Operationen auf diesen verlinkten Bauteilen auszuführen. So könnte man bspw. mit weiteren Get-Anfragen eine rekursive Auflösung der Stückliste durchführen.

5.1.4 Testen von Edit-Zugriffen

Um die weitere Funktionalität zu testen, wird dieses mal eine POST Anfrage an das selbe Bauteil gesendet. Dabei soll der Name des Bauteils von *Rotor Frame* zu *Motor Frame* geändert werden. Bei dieser Anfrage wird ein Payload innerhalb der Nachricht verlangt, indem im JSON-LD Format die zu verändernden Attribut und Wert Paare angegeben werden. Der Payload der Nachricht ist in **Listing 13** zu sehen.

Listing 13: Payload einer POST-Anfrage an ein Part.

```
{
    "@context" : {
        "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/"
    },
    "@id" : "http://aras.thomas-psota.de:6112/Part/P1028",
    "aras:Name" : "Motor Frame",
}
```

Der Erfolg der Anfrage wird vom Wrapper mit einem Status Code signalisiert. Ein Blick in die Oberfläche der Aras Innovator PDM Instanz zeigt, dass das Bauteil erfolgreich verändert

wurde, wie in dem Screenshot in **Abb. 21** zu sehen ist. Durch die erfolgreiche Änderung des Bauteils wurde das Systemattribut Generation von 1 auf 2 gesetzt. Der Name wurde mit dem gewünschten Wert befüllt und die Historie zeigt den Wrapper als letzten Veränderer des Bauteils an.

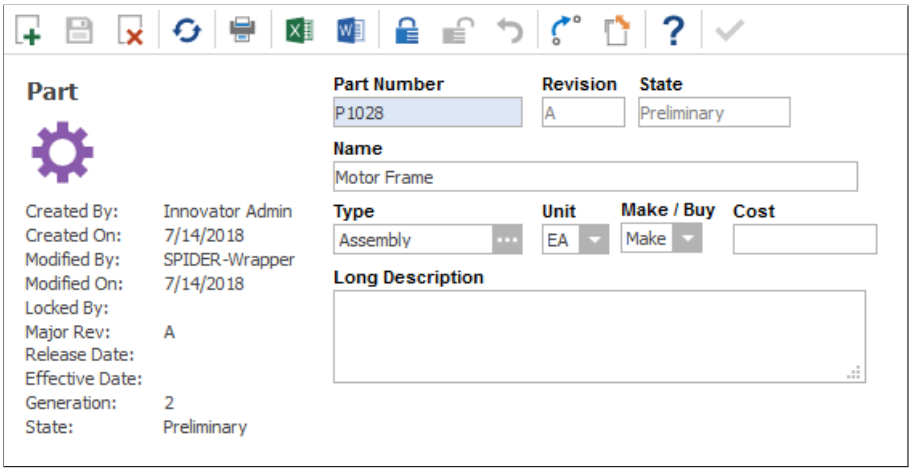

Abbildung 21: Screenshot des veränderten Bauteils in der Aras Innovator PDM Instanz.

5.1.5 Testen von Create-Zugriffen

Um zu Testen ob sich neue Bauteile erstellen lassen, wird eine PUT-Anfrage an die Resource *http://aras.thomas-psota.de:6112/Part* geschickt. Diese enthält als Payload eine ausgefüllte Instanz der Klasse im JSON-LD Format. Das für diesen Test verwendete Dokument ist in **Listing 14** zu sehen.

Listing 14: Payload einer PUT-Anfrage zur Erstellung eines neuen Bauteils in der Aras Innovator PDM Instanz

```
{
    "@context" : {
        "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/"
    },
    "@type" : "aras:Part",
    "aras:Name" : "Test Part",
    "aras:PartNumber" : "T001",
    "aras:Type" : "Component"
}
```

Der Erfolg der Operation wird durch einen Status Code signalisiert. Eine Überprüfung in der Oberfläche der Aras Innovator PDM Instanz bestätigt, dass ein neues Teil mit den gewünschten Werten erstellt wurde, wie im Screenshot in **Abb. 22** zu sehen ist. Dort wurde auch der Wrapper als Ersteller des Bauteils akkreditiert.

5.1.6 Testen von Delete-Zugriffen

Um die Funktionalität der Delete-Zugriffe zu testen, wird das vorher erstellte Bauteil mit der Teilenummer T001 versucht zu löschen. Dafür wird eine HTTP-Anfrage vom Typen DELETE auf die Resource *http://aras.thomas-psota.de/Part/T001* gesendet.

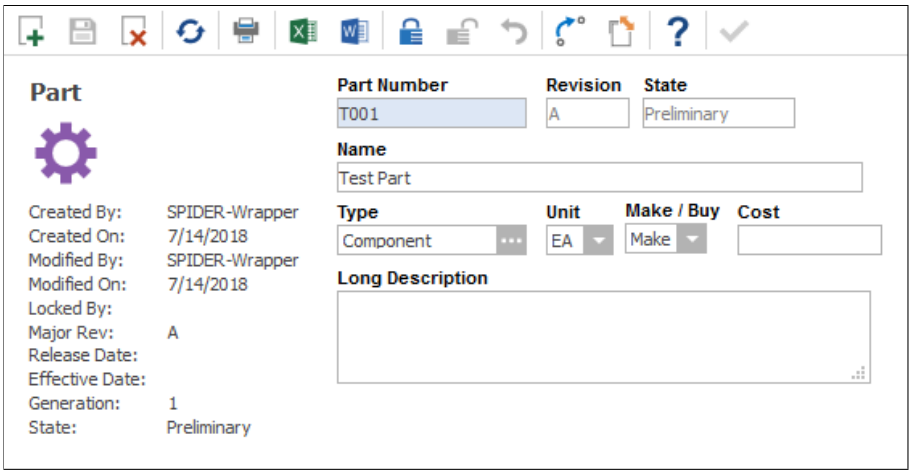

Abbildung 22: Screenshot eines durch SPIDER erstellten Bauteils in der Aras Innovator PDM Instanz.

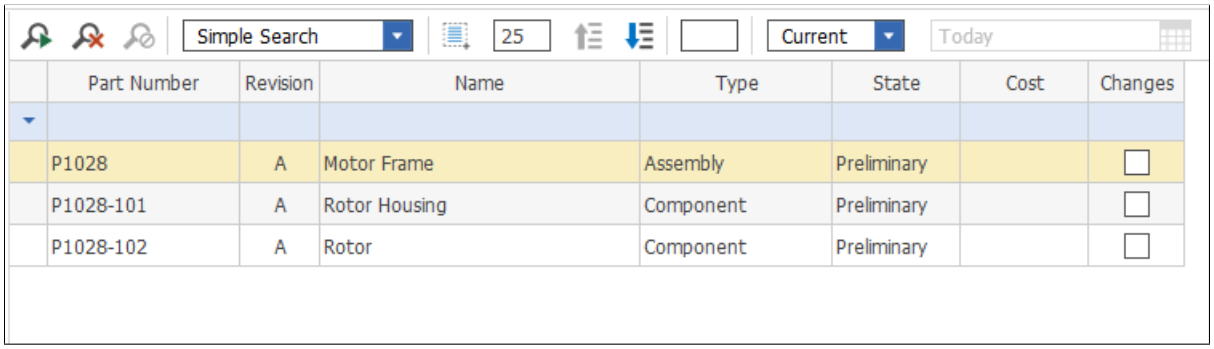

Abbildung 23: Screenshot der Auflistung aller im System befindlichen Bauteile der Aras Innovator PDM Instanz.

Die erfolgreiche Löschung wird durch einen Status Code signalisiert. Zur Überprüfung wird in der Oberfläche der Aras Innovator PDM Instanz eine Auflistung aller im System befindlichen Bauteile durchgeführt. Diese Liste ist im Screenshot in **Abb. 23** abgebildet. Wie darauf zu erkennen ist, taucht das Bauteil mit der Teilenummer T001 nicht in der Liste auf. Demnach hat eine Löschung des Bauteils erfolgreich stattgefunden.

5.2 Anbindung eines CASE Tools

In diesem Abschnitt soll die Anbindung eines CASE-Tools an SPIDER erprobt werden. Aufgrund seiner weiten Verbreitung in der Industrie und der guten Dokumentation wird als anzubindendes CASE-Tool Git gewählt. Die Hauptaufgabe eines CASE-Tools im PLM Umfeld ist die Verwaltung von Software-Fragmenten, darum soll versucht werden ein Git Repository mit den Metadaten all seiner Commits an SPIDER anzubinden.

Da leider keine Testdaten aus dem PLM Umfeld zur Verfügung gestellt werden konnten, wurde stattdessen das Repository der Open-Source Software *vscode*[26] verwendet. Gewählt wurde dieses Repository, da es mit einer Anzahl von über 30.000 Commits eine gute Grundlage für Performance Tests der SPIDER Architektur liefert. Dazu wurde eine lokale Kopie des Repositories auf einem Server erstellt und dieses für die weitere Anbindung verwendet. Für das Stellen und die Zeitmessung der HTTP-Anfragen wird das freie Open-Source Programm *curl* verwendet.

5.2.1 Anbindung der Datenquelle

Für die Integration sollen die Commits des Repositories zur Verfügung gestellt. Folgende Metadaten eines Commits sollen dabei verwendet werden:

- Der SHA-1 Code des Commits.
- Die Nachricht des Commits.
- Die E-Mail Adresse des Autors.
- Die Vorgänger-Commits, falls existent.

Analog zu der Vorgehensweise bei der Anbindung des PDM Systems, wurde auch hier wieder eine Datenquelle sowie ihr Datenmodell unter Verwendung des SPIDER-Core erstellt. Die Konfiguration der Klasse ist dabei dem Screenshot in **Abb. 24** zu entnehmen.

Als identifizierendes Attribut wurde der SHA-1 Code des Commits gewählt, da dieser semantisch auch von Git für diesen Zweck verwendet wird. [19] Vorgänger-Commits wurden als Liste definiert, da es mehrere geben kann. Da in diesem Test nur lesende Zugriffe getätigt werden, wurden Schreibzugriffe auf der Klasse deaktiviert.

Für das Auslesen der Informationen aus dem Git Repository wird die C# Library *LibGit2Sharp*[25] verwendet. Mit dem Generator wurde der entsprechende Stub für den Wrapper erzeugt und mit den Logiken zum Auslesen des Git Repositories ausgefüllt. Der fertige Code für den Wrapper wurde dieser Arbeit in **Anhang E** beigefügt.

5.2.2 Inbetriebnahme der Anbindung

Der fertige Wrapper wurde auf dem gleichen Server, auf dem sich auch das lokale Repository befindet, aufgespielt. Er wurde unter der URL `http://git.thomas-psota.de:2810` verfügbar gemacht.

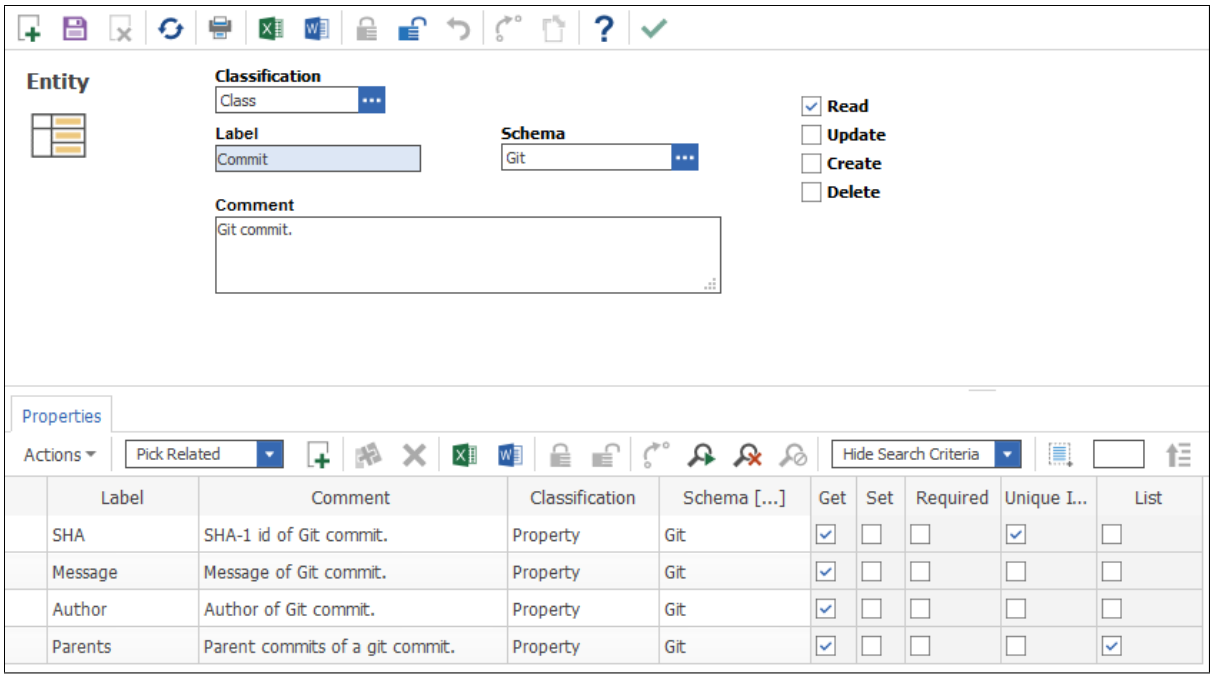

Abbildung 24: Konfiguration der Klasse Commit innerhalb des SPIDER-Core.

5.2.3 Performance-Analyse der Get-Zugriffe

Unter der Resource `http://git.thomas-psota.de:2810/Commit` ist eine Auflistung aller Commits des Repositories abrufbar. Ein Ausschnitt dieser Resultate ist in **Listing 15** zu finden.

Listing 15: Ausschnitt der Auflistung aller Commits im angebundenen Git Repository.

```
1
2  {
3      "@context" : {
4              "git": "http://spider.thomas-psota.de:8080/Schema/git/"
5      },
6      "@graph" :
7      [
8          {
9              "@id" : "http://git.thomas-psota.de:2810/Commit/
                   a846f437d8a9888932e01c02a741792cdb1c11d7",
10             "@type" : "git:commit"
11         },
12         {
13             "@id" : "http://git.thomas-psota.de:2810/Commit/
                   fbd432b9abbcae808b88229e83ba9f88f64d86d3",
14             "@type" : "git:commit"
15         },
16         {
17             "@id" : "http://git.thomas-psota.de:2810/Commit/8
                   b9495d494ab5ec84d8909ffb282908cfcb07bf9",
18             "@type" : "git:commit"
19         },
20         {
```

```
            "@id" : "http://git.thomas-psota.de:2810/Commit/
                c150d0d42d7385c9aa32ffbff99573ada4916f4e",
            "@type" : "git:commit"
        },
        {
            "@id" : "http://git.thomas-psota.de:2810/Commit/
                e89ed6eb3a82fc3f47b362fcef0455d5d35261f1",
            "@type" : "git:commit"
        },
        {
            "@id" : "http://git.thomas-psota.de:2810/Commit/
                f9e0f8d5ce5e0f3a5cfcf30669bae36e173f9933",
            "@type" : "git:commit"
        },
        {
            "@id" : "http://git.thomas-psota.de:2810/Commit/
                eaa535ff3e3fafe37b1f30e546023b04b45d02a1",
            "@type" : "git:commit"
        },
        ...
    ]
}
```

Die durchschnittliche gemessene Dauer bis zum Erhalte der Antwort auf die Anfrage ist in **Tabelle 16** aufgeführt. Dabei wurden die Werte von zehn gemessenen Anfragen aggregiert. Die Resultate dieser einzelnen Messungen sind in **Anhang F** angefügt.

Ausführungsschritt	Dauer
Senden der Anfrage	< 1ms
Bearbeitung der Anfrage	~5s
Übertragung der Antwort	< 1s

Tabelle 16: Durchschnittliche Bearbeitungszeit eines Get-Zugriffs auf die Collection Commit.

Diesen Resultaten ist zu entnehmen, dass die Dauer für die Übertragungen von Anfrage und Nachricht zu vernachlässigen ist. Die Bearbeitung der Anfrage im Wrapper nimmt den Großteil der gemessenen Dauer ein. Demnach benötigt der Wrapper im Durchschnitt fünf Sekunden um durch alle Commits des Repositories zu iterieren und diese in seiner Antwort zusammenzufassen.

Anschließend wurde eine Messung der Zeit für die Bearbeitung einer Anfrage für den Get-Zugriff auf einen einzelnen Commit gemessen. Es wurden Anfragen auf zehn verschiedene Commits ausgeführt und der Durchschnitt der Messergebnisse in **Tabelle 17** zusammengefasst.

Übertragungszeiten von Anfrage und Antwort sind demnach wieder zu vernachlässigen. Zu sehen ist, dass der Wrapper für das Auffinden und Auslesen der Metadaten eines Commits ungefähr eine halbe Sekunde benötigt. Würde eine Client-Anwendung versuchen über SPIDER Informationen über jeden einzelnen Commit im Repository abzufragen, würde dies, bei über

Ausführungsschritt	Dauer
Senden der Anfrage	< 1ms
Bearbeitung der Anfrage	~0.48s
Übertragung der Antwort	< 1ms

Tabelle 17: Durchschnittliche Bearbeitungszeit eines Get-Zugriffs auf einen Commit.

30.000 Commits im Repository, eine Bearbeitungszeit von über vier Stunden bedeuten. Demnach wird hier ein klares Bottleneck von SPIDER lokalisiert.

5.3 Änderungen an den Datenquellen

In diesem Abschnitt soll SPIDER auf seine Möglichkeiten untersucht werden, auf Änderungen der Datenquellen zu reagieren. Im ersten Szenario soll untersucht werden, welche Auswirkungen eine Änderung am Datenmodell des Quellsystems auf SPIDER hat. Im zweiten Szenario soll eine angebundene Datenquelle ausgetauscht und durch ein anderes System ersetzt werden, bei einem gleich bleibendem Datenmodell.

5.3.1 Änderungen am Datenmodell

In diesem Versuch soll eine Änderung am PDM System von der in Kapitel 5.1 konfigurierten SPIDER Installation vorgenommen werden.

In diesem fiktiven Szenario wurde ein Update des PDM Systems im Unternehmen installiert, wodurch dessen Datenmodell erweitert wurde. Um diese Änderung zu simulieren, soll das Schema des Bauteils von Aras Innovator um das Attribut Generation erweitert werden.

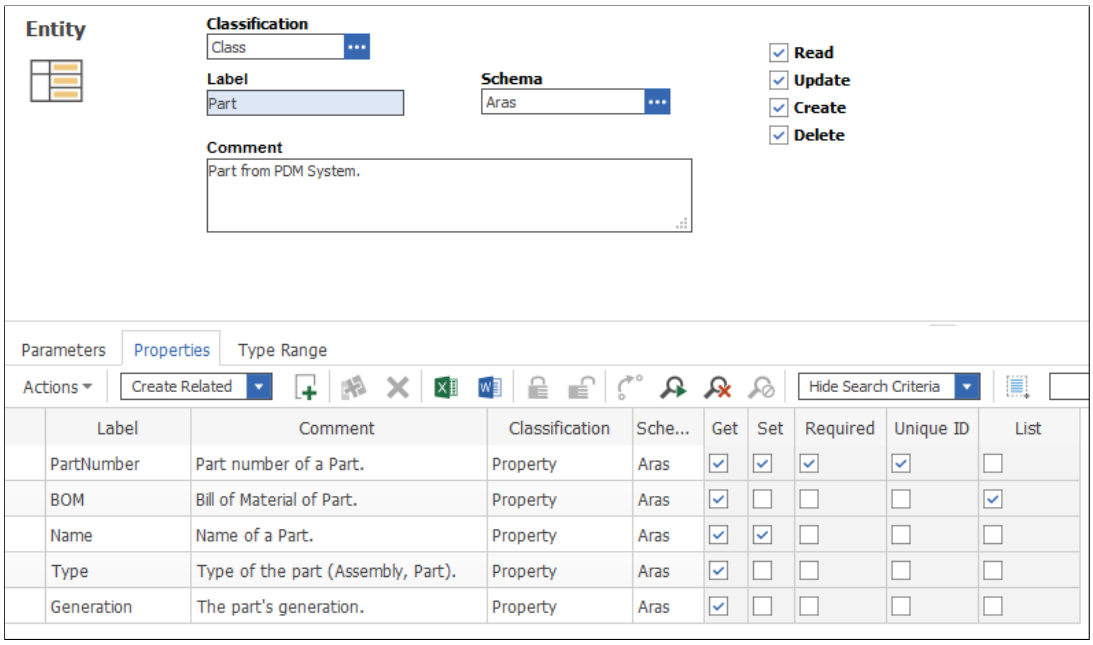

Abbildung 25: Angepasste Klassenkonfiguration für Part in SPIDER-Core.

Diese Änderung lässt sich zunächst ohne größere Schwierigkeiten in der Oberfläche des SPIDER-Core vornehmen. Die angepasste Konfiguration der Klasse Part ist in **Abb. 25** zu sehen.

Das veränderte Schema lässt sich weiterhin an der von SPIDER-Core zur Verfügung gestellten URL abrufen. Da der Wrapper an das neue Attribut angepasst werden muss, wird zunächst das neue Schema benutzt, um den Wrapper für das angepasste Datenmodell zu generieren.

Da Änderungen am Datenmodell bei der Generierung keine Auswirkungen auf die *Handler* Klasse haben, lässt sich die alte, bereits vorhandene *Handler* Klasse weiterverwenden. In dieser muss dennoch der Zugriff auf die Datenquelle angepasst werden. Die benötigten Änderungen für den Get-Zugriff sind dabei in Listing **Listing 16** rot markiert worden. Demnach musste in

diesem Fall nur eine Zeile Code hinzugefügt werden, nämlich nur eine Anpassung der proprietären Logik der Datenquelle. Ähnliche Änderungen müssten dabei auch in den übrigen Methoden der *Handler* Klasse für das Bauteil stattfinden.

Listing 16: Angepasste Methode *Get* der *Handler* Klasse für Parts. Durchgeführte Änderungen sind rot markiert.

```
// Return object of given ID
public static Object.Part Get(String ID)
{
    Object.Part part = new Object.Part();

    var partQry = Global.aras.inn.newItem("Part", "get");
    partQry.setProperty("item_number", ID);
    var arasPart = partQry.apply();

    part.PartNumber = arasPart.getProperty("item_number");
    part.Name = arasPart.getProperty("name");
    part.Type = arasPart.getProperty("classification");
    part.Generation = arasPart.getProperty("generation");

    arasPart.fetchRelationships("Part BOM");
    var part_bom = arasPart.getRelationships("Part BOM");
    for (int i = 0; i < part_bom.getItemCount(); ++i)
    {
            var component = part_bom.getItemByIndex(i).getRelatedItem();
            var componentPartNo = component.getProperty("item_number");

            part.BOM.Add(new Object.Part(componentPartNo));
    }

    return part;
}
```

Eine Inbetriebnahme des angepassten Wrappers sowie eine Get-Anfrage auf das Bauteil mit der Nummer liefern das Resultat aus **Listing 17**. Die der Antwort neu hinzugefügte Zeile mit dem neuen Attribut wurde rot markiert.

Damit wurde eine Änderung des Datenmodells erfolgreich in SPIDER durchgeführt.

Listing 17: Get-Anfrage auf das Bauteil mit der Teilenummer P1028 über den angepassten Wrapper. Der neue Teil der Antwort wurde in rot markiert.

```
{
    "@context" : {
        "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/"
    },
    "@id" : "http://aras.thomas-psota.de:6112/Part/P1028",
    "@type" : "aras:Part",
    "aras:BOM" :
    [
        {
            "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-101",
            "@type" : "aras:Part"
```

```
        },
        {
            "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-102",
            "@type" : "aras:Part"
        }
    ],
    "aras:Name" : "Motor Frame",
    "aras:PartNumber" : "P1028",
    "aras:Type" : "Assembly",
    "aras:Generation" : "2",
}
```

5.3.2 Änderungen an den Datenquellen

In diesem fiktiven Szenario soll ein Austauschen des PDM Systems simuliert werden. Eine Migration der Daten kann dabei vorausgesetzt werden. Dabei soll das momentane Datenmodell unangetastet bleiben. Als neues Quellsystem soll dabei eine SQL Datenbank verwendet werden.

Da die PDM Instanz von Aras Innovator bereits einen Microsoft SQL-Server als Datenbank verwendet, wird dieser für diesen Test verwendet. Da keine Änderungen am Datenmodell durchgeführt werden, kann die momentane Konfiguration innerhalb des SPIDER-Core weiter verwendet werden. Demnach müssen auch keine neuen Dateien für den Wrapper generiert werden. Um diesen für die neue Datenquelle anzupassen, genügt es die Logiken für die Zugriffe ins Quellsystem in den Methoden der *Handler* Klasse anzupassen. Dies entspricht einem Austausch des *Handler*-Layers in der Architektur des Wrappers. Die neue Logik für den Zugriff in die SQL Datenbank befindet sich in Listing **Listing 18**.

Listing 18: Angepasste Methode *Get* der *Handler* Klasse für Parts. Durchgeführte Änderungen sind rot markiert.

```
// Return object of given ID
public static Object.Part Get(String ID)
{
    Object.Part part = new Object.Part();

    cmd.CommandText =
        @"SELECT item_number,name,classification,generation
          FROM innovator.Part
          WHERE item_number = '" + ID + "'";
    cmd.CommandType = CommandType.Text;
    cmd.Connection = Global.SqlConnection;

    SqlDataReader reader;
    reader = cmd.ExecuteReader();

    reader.Read();
    part.PartNumber = reader.GetString(0);
    part.Name = reader.GetString(1);
    part.Type = reader.GetString(2);
    part.Generation = reader.GetString(3);
```

```
21
22      reader.Close();
23
24      SqlCommand bomCmd = new SqlCommand();
25      bomCmd.CommandText =
26          @"SELECT item_number FROM innovator.PART WHERE id IN
27            (SELECT related_id FROM innovator.PART_BOM bom
28            JOIN innovator.part p ON p.id = bom.SOURCE_ID)";
29      bomCmd.CommandType = CommandType.Text;
30      bomCmd.Connection = Global.SqlConnection;
31
32      SqlDataReader bomReader;
33      bomReader = bomCmd.ExecuteReader();
34
35      while (bomReader.Read()) {
36          part.BOM.Add(bomReader.GetString(0));
37      };
38
39      bomReader.Close();
40
41      return part;
42  }
```

Um den angepassten Wrapper zu testen wurde eine Get-Anfrage an das Bauteil mit der Teilenummer P1028 gesendet. Die erhaltene Antwort des angepassten Wrappers ist in Listing **Listing 19** zu sehen. Die Antwort des SQL Wrappers ist dabei unverändert zu der des Aras Innovator Wrappers. Dies entspricht den Erwartungen, da das Datenmodell unverändert blieb und lediglich Änderungen an dem unterliegenden Quellsystem stattfanden.

Listing 19: Antwort des SQL Wrappers auf eine Get-Anfrage.

```
1  {
2      "@context" : {
3          "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/"
4      },
5      "@id" : "http://aras.thomas-psota.de:6112/Part/P1028",
6      "@type" : "aras:Part",
7      "aras:BOM" :
8      [
9          {
10             "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-101",
11             "@type" : "aras:Part"
12         },
13         {
14             "@id" : "http://aras.thomas-psota.de:6112/Part/P1028-102",
15             "@type" : "aras:Part"
16         }
17     ],
18     "aras:Name" : "Motor Frame",
19     "aras:PartNumber" : "P1028",
20     "aras:Type" : "Assembly",
21     "aras:Generation" : "2",
22 }
```

5.4 Auswertung der Ergebnisse

In diesem Abschnitt findet eine Auswertung der gesammelten Ergebnisse statt. So konnte im ersten Szenario erfolgreich ein PDM System an SPIDER angeschlossen werden. Eine Konfiguration der Datenquelle und ihrem Datenmodell ließ sich dabei intuitiv über die web-Oberfläche des SPIDER-Core vornehmen.

Durch das erzeugte Schema der Klasse *Part* konnte mit dem Generator der Stub für einen Wrapper erzeugt werden. Die restliche Implementierung des Wrappers bestand dabei nur aus den für die Verbindung zur Datenquelle benötigten proprietären Logiken. Demnach müsste ein potenzieller Programmierer über keine Interna über die vom Wrapper verwendete Logik der Datentransformation oder REST-Schnittstelle verfügen, um einen funktionierenden Wrapper für eine Datenquelle zu erstellen.

Ebenfalls konnte die Funktionalität des Wrappers anhand von Testdaten geprüft werden. Dieser hat alle erforderlichen Interfaces der REST-Schnittstelle erfüllt sowie die erforderlichen Operationen innerhalb der Datenquelle ausgeführt.

Das Kriterium der einfachen Anbindung von neuen Datenquellen an SPIDER sowie der unkomplizierten Erstellung von Wrappern ließe sich damit als erfüllt betrachten.

In einem zweiten Szenario wurde mit Git eine weitere Datenquelle an SPIDER angeschlossen. Wieder konnte über die Web-Oberfläche das Schema definiert sowie die Registrierung vorgenommen werden, und ein Wrapper aus dem generierten Schema erstellen lassen.

Eine Performance Analyse der Get-Anfragen offenbarte dabei eine der Schwachstellen von SPIDER. Demnach existiert ein Bottleneck bei der Verarbeitung von mehreren aufeinanderfolgenden Anfragen. So ist die Zeit zur Bearbeitung einer einzelnen Anfrage mit, in diesem Fall, einer halben Sekunde zwar noch in einem akzeptablem Bereich für eine einzelne Anfrage. Jedoch macht sich diese Länge schnell bemerkbar wenn man Anfragen nach mehreren Objekten senden will, wie bspw. bei der Auflösung einer Stückliste oder dem Durchlaufen der Auflistung einer Collection.

Dies ist zum Einen auf die Tatsache zurück zu führen, dass die Implementierung des Wrappers kein Multi-Threading verwendet. Eine mögliche Implementierung des Wrappers mit Threading würde es zumindest ermöglichen mehrere Anfragen gleichzeitig zu beantworten.

Eine andere Lösung für das Problem, wäre die Möglichkeit verschachtelte Anfragen zu senden. Somit könnten bspw. mehrere IDs in einer Anfrage angefragt werden. Denn wie am Beispiel der Auflistung zu sehen war, ist es für den Wrapper schneller die Daten der Datenquelle in einer Iteration auszulesen, als dies für jedes in der Datenquelle enthaltene Objekte einzeln zu tun.

In einem dritten Test wurde eine Änderung am Datenmodell einer Datenquelle erprobt. Um diese Änderung in SPIDER umzusetzen musste zunächst das im SPIDER-Core definierte Schema der Datenquelle angepasst werden und anschließend ein neuer Wrapper erzeugt

werden. Dabei konnte der *Handler* des alten Wrappers zunächst weiter verwendet werden, da die erzeugten Änderungen nur die im Wrapper verwendeten Schichten des Servers sowie der Datentransformation berührten. Nach einer Anpassung des *Handler*-Codes auf das neue Datenmodell der Datenquelle konnte der Wrapper wieder in Betrieb genommen werden.

Damit konnte gezeigt werden, dass SPIDER auf Änderungen von Datenmodellen reagieren kann.

Im vierten Test wurde das Szenario des Wechsels einer bereits an SPIDER angebundenen Datenquelle durchgespielt. Dies erforderte dabei nur eine Anpassung des *Handler*-Codes des bereits existierenden Wrappers. Eine Get-Anfrage lieferte dabei das identische Resultat wie die alte Datenquelle. Demnach würde ein Anwender den Wechsel der Datenquelle gar nicht realisieren.

Somit konnte gezeigt werden, dass solange das Datenmodell der Datenquelle gleich bleibt, ein Austauschen der Datenquelle keine nennenswerten Probleme für SPIDER erzeugt.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde mit SPIDER eine generische Integrationsplattform für die Anbindung verschiedener Anwendungen innerhalb des Produktentstehungprozesses eines Unternehmens geschaffen. Dabei wurden zunächst die Grundlagen des PLM sowie der Datenintegration betrachtet sowie ein kurzer Überblick über verwandte Arbeiten erstellt.

Ausgehend von diesen Grundlagen konnte ein Use Case erarbeitet werden, welcher für die weitere Entwicklung des Systems verwendet wurde. Dabei wurden Anforderungen sowie Qualitätskriterien abgeleitet. Zusätzlich fand eine Betrachtung ausgewählter Quellsysteme statt, durch welche die Anforderungen für die semantische Beschreibung der Datenquellen abgeleitet werden konnten. Diese wurden anschließend in einer erstellten Ontologie festgehalten.

Ausgehend aus den Anforderungen ließ sich eine generische Architektur für die Plattform erstellen. In dieser konnten die Komponenten des SPIDER-Core sowie der Wrapper identifiziert werden. Beim Design der Wrapper fand eine Analyse ihrer Variabilität statt. Dadurch konnten diese als Core Assets realisiert werden und zusätzlich ein Generator implementiert werden, der Wrapper-Stubs anhand der semantischen Beschreibung einer Datenquelle erstellen kann.

Ebenso ließ sich eine Implementierung des SPIDER-Core mittels der Middleware Aras Innovator realisieren, wodurch eine intuitive Konfiguration der von SPIDER verwalteten Datenquellen und Datenmodelle möglich wurde.

Die erstellten Implementierungen wurden dabei anhand von möglichen Szenarien für die Anwendung von SPIDER evaluiert.

6.1 Ausblicke

In seiner jetzigen Form wurde mit SPIDER nur der Grundsatz für eine durchgängige Ausführung von PLM gelegt. Die jetzigen Operationen von SPIDER beschäftigen sich nur mit den lokalen Daten einer einzigen Quelle. Um eine vollständige Unterstützung für Abläufe im PLM zu bilden, müssen Interaktionen zwischen den angebunden Quellsystemen in SPIDER stattfinden. Demnach muss der jetzige Ansatz von SPIDER, welcher sich vorerst nur auf Produktdaten beschränkt, um die Berücksichtigung von Prozessdaten und deren Informationen erweitert werden.

Durch diese Erweiterung von Prozessdaten würde eine Entwicklung von SPIDER zu SP^{2}IDER stattfinden, also zu einem *Semantic Product/Process Information and Digitized Engineering Repository*.

Dabei wäre der nächste Schritt, um die von SPIDER zur Verfügung gestellten semantischen Informationen zu nutzen, der Aufbau der semantischen Netze des Unternehmens. Dafür wäre eine Datenfusion mit den angebundenen Quellsystemen erforderlich. Dabei treten bis jetzt noch ungelöste Probleme auf, wie bspw. das Erkennen von semantisch gleichen Bauteilen in verschiedenen Systemen. Ebenso müssten Möglichkeiten festgestellt werden, Medienbrüche

zwischen verschiedenen Systemen aufzudecken und diese zu überbrücken. Auch müssten zusätzliche Möglichkeiten zur Verfügung gestellt werden um diese semantischen Netze verwalten zu können.

Da mit SPIDER eine Abstraktionsschicht auf die einzelnen Anwendungen eines Unternehmens gelegt wurde, wäre es nun auch denkbar generische Anwendungen zu schreiben, die diese Schicht als Interface verwenden, um generische PLM Lösungen zu entwickeln, die nicht von den eingesetzten Anwendungen eines Unternehmens abhängig sind.

Und auch das in der Evaluierung festgestellte Bottleneck bei der Anfragebearbeitung ist ebenfalls noch eine ausstehende Verbesserungsmöglichkeit von SPIDER. Eine weitere Verbesserungsmöglichkeit ließe sich ebenfalls bei der Ortstransparenz der angebundenen Datenquellen erzielen. Bis jetzt sind Anwender darauf angewiesen die genaue Position des Wrappers der Datenquelle im Netzwerk zu kennen um Anfragen an diese zu senden. Eine Möglichkeit wäre Mechanismen zur Indirektion der Anfragen zu erstellen, die eine automatische Auflösung der Netzwerkadresse vornehmen.

Literatur

[1] World Wide Web Consortium (w3c). *A JSON-based Serialization for Linked Data*. 2014. URL: https://www.w3.org/TR/2014/REC-json-ld-20140116/ (besucht am 05. 05. 2018).

[2] World Wide Web Consortium (w3c). *JSON-LD 1.0 Processing Algorithms and API*. 2014. URL: https://www.w3.org/TR/2014/REC-json-ld-api-20140116/ (besucht am 05. 05. 2018).

[3] World Wide Web Consortium (w3c). *RDF 1.1 Primer*. 2014. URL: https://www.w3.org/TR/rdf11-primer/ (besucht am 05. 05. 2018).

[4] World Wide Web Consortium (w3c). *RDF 1.1 XML Syntax*. 2014. URL: https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/ (besucht am 05. 05. 2018).

[5] World Wide Web Consortium (w3c). *RDF Schema 1.1*. 2014. URL: https://www.w3.org/TR/2014/REC-rdf-schema-20140225/ (besucht am 05. 05. 2018).

[6] *About Schema.org*. URL: https://schema.org/docs/about.html (besucht am 12. 05. 2018).

[7] Michael Abramovici, Jens Christian Göbel und Hoang Bao Dang. "Semantic data management for the development and continuous reconfiguration of smart products and systems". In: *CIRP Annals-Manufacturing Technology* 65.1 (2016), S. 185–188.

[8] Michael Abramovici u. a. "A Semantic Information Retrieval Framework within the Scope of IPS2-PLM". In: *Procedia CIRP* 47 (2016), S. 294–299.

[9] Reiner Anderl u. a. *Smart Engineering: Interdisziplinäre Produktentstehung*. acatech Diskussion. Berlin: Springer, 2012.

[10] Grigoris Antoniou und Frank van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.

[11] Volker Arnold u. a. *Product Lifecycle Management beherrschen: Ein Anwenderhandbuch für den Mittelstand*. 2. Aufl. München: Springer, 2011.

[12] *Boost C++ Libraries*. 2018. URL: https://www.boost.org/ (besucht am 08. 07. 2018).

[13] Aras Corp. *Aras Innovator 11 - Configuring Solutions Guide*. 2018.

[14] Stefan Deßloch u. a. "Information Integration - Goals and Challenges". In: *Datenbank-Spektrum* 6 (2003), S. 7–13.

[15] Martin Eigner, Walter Koch und Christian Muggeo. *Modellbasierter Entwicklungsprozess cybertronischer Systeme*. Springer, 2017.

[16] Martin Eigner, Daniil Roubanov und Radoslav Zafirov. *Modellbasierte Virtuelle Produktentwicklung*. Berlin: Springer, 2014.

[17] Martin Eigner und Ralph Stelzer. *Product Lifecycle Management: Ein Leitfaden für Product Development und Life Cycle Management*. 2. Aufl. VDI-Buch. Dordrecht: Springer, 2009.

[18] Lars Geyer und Martin Becker. "On the Influence of Variabilities on the Application-Engineering Process of a Product Family". In: *Software Product Lines*. Lecture Notes in Computer Science SPLC 2002.2379 (2002), S. 1–14.

[19] Git. *git Documentation*. 2018. URL: https://git-scm.com/docs/git (besucht am 08. 07. 2018).

[20] GitHub. *JsonCpp*. 2018. URL: https://github.com/open-source-parsers/jsoncpp (besucht am 15. 06. 2018).

[21] IDG Business Media GmbH. *Studie Analytics Readiness 2016*. Studie. 2016.

[22] *Introducing JSON*. URL: https://www.json.org/ (besucht am 05. 05. 2018).

[23] Michael Küpper. *Engineering Cockpit: Enabling Smart Engineering @ Schaeffler*. Conference Slides. IBM Symposium Systèmes & ALM PARIS. IBM Germany, Paris, 2017.

[24] Ulf Leser und Felix Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt.verlag, 2007.

[25] *LibGit2Sharp Hitchhikers Guide to Git*. 2018. URL: https://github.com/libgit2/libgit2sharp/wiki/LibGit2Sharp-Hitchhiker%27s-Guide-to-Git (besucht am 08. 07. 2018).

[26] Microsoft. *vscode*. 2018. URL: https://github.com/Microsoft/vscode (besucht am 08. 07. 2018).

[27] Microsoft MSDN. *C++ Calling Conventions*. 2018. URL: https://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx (besucht am 08. 07. 2018).

[28] Microsoft MSDN. *Creating and Using a Dynamic Link Library (C++)*. 2018. URL: https://msdn.microsoft.com/en-us/library/ms235636.aspx (besucht am 15. 06. 2018).

[29] Microsoft MSDN. *Dynamic-Link Libraries*. 2018. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589%28v=vs.85%29.aspx (besucht am 15. 06. 2018).

[30] Paul Müller. *Service-oriented Computing, Summer Term 2016, Slide Set 3 – Communication*. Lecture Slides. TU Kaiserslautern. 2017.

[31] OMG. *What is SysML?* 2018. URL: http://www.omgsysml.org/what-is-sysml.htm (besucht am 21. 04. 2018).

[32] Mark Richards. *Software Architecture Patterns: Understanding Common Architecture Patterns and when to Use Them*. Sebastopol, California: O'Reilly Media, Incorporated, 2015.

[33] Ian Sommerville. *Software Engineering*. 9. Aufl. Pearson, 2012.

[34] *What is OSLC? - OSLC Primer - Open Services for Lifecycle Collaboration*. URL: http://open-services.net/resources/tutorials/oslc-primer/what-is-oslc (besucht am 05. 05. 2018).

[35] *What is SWIG*. 2018. URL: http://www.swig.org/exec.html (besucht am 15. 06. 2018).

[36] *Why AP242? - ePLM Interoperability*. URL: http://www.ap242.org/why-ap242 (besucht am 10. 06. 2018).

[37] Wikipedia. *Programmbibliothek*. 2018. URL: https://de.wikipedia.org/wiki/Programmbibliothek (besucht am 28. 06. 2018).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A Object.Template.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using SPIDER;

using @@PROJECT@@.Internal;

namespace @@PROJECT@@.Object
{
    class @@OBJECT@@
    {
        public static Context context;
        public const String TypeID = "@@OBJECT_ID@@";
        ## FOREACH @@PROPERTY@@
        ## public @@PROPERTY_TYPE@@ @@PROPERTY@@ { get; set; }
        @@SINGLE_PROPERTIES@@

        ## FOREACH @@LIST@@
        ## public List<@@PROPERTY_TYPE@@> @@LIST@@ = new List<string>();
        @@LIST_PROPERTIES@@

        public @@OBJECT@@()
        {

        }

        public @@OBJECT@@( String id)
        {
            @@OBJECT_UNIQUE_ID@@ = id;
        }

        public String GetID()
        {
            return @@OBJECT_UNIQUE_ID@@;
        }

        public String CreateNodeID(RuntimeInfo info)
        {
            return info.HostAddress + "/@@OBJECT@@/" + GetID();
        }

        public Node toRefNode(RuntimeInfo info)
        {
            Node node = new Node();

            node.setID(this.CreateNodeID(info));
            node.setType(TypeID);

```

```
51              return node;
52          }
53
54          private Graph serialize(RuntimeInfo info)
55          {
56              ContextBuilder contextBuilder = new ContextBuilder();
57              Context context = contextBuilder.Build();
58
59              Node node = new Node();
60              String term;
61
62              node.setID(this.CreateNodeID(info));
63              node.setType(Program_Part.TypeID);
64
65              @@SERIALIZE_SINGLE_PROPERTIES@@
66              @@TEMPLATE@@
67              // Serialize @@PROPERTY_LABEL@@
68              if (@@PROPERTY_LABEL@@ != null)
69              {
70                  term = context.compactIRI("@@PROPERTY_ID@@");
71                  @@PROPERTY_SERIALIZER@@
72              };
73              @@TEMPLATE@@
74
75              @@SERIALIZE_LIST_PROPERTIES@@
76              @@TEMPLATE@@
77              // Serialize @@PROPERTY_LABEL@@
78              term = context.compactIRI("@@PROPERTY_ID@@");
79              foreach (var @@PROPERTY_LABEL@@ in @@PROPERTY_LABEL@@)
80              {
81                  @@PROPERTY_SERIALIZER@@
82              }
83              @@TEMPLATE@@
84
85              Graph graph = new Graph(context);
86              graph.AddNode(node);
87
88              return graph;
89          }
90
91          private static @@OBJECT@@ fromRequest(Request request)
92          {
93              var graph = request.getGraph();
94              var context = graph.getContext();
95              var node = graph.GetNode(0);
96
97              @@OBJECT@@ obj = new @@OBJECT@@();
98              String value;
99
100             @@DESERIALIZE_PROPERTIES@@
101             @@TEMPLATE@@
102             // Deserialize @@PROPERTY_LABEL@@
103             value = node.getValue(context.compactIRI("@@PROPERTY_ID@@"));
```

```
104             if (!String.IsNullOrEmpty(value))
105                 obj.@@PROPERTY_LABEL@@ = value;
106 
107             @@TEMPLATE@@
108 
109             return obj;
110         }
111 
112                 @@UPDATE_ALLOWED@@
113         private static bool validatePostRequest(Request request, RuntimeInfo info)
114         {
115             var graph = request.getGraph();
116 
117             if (graph == null || graph.Count() == 0)
118             {
119                 request.SendError("Not a valid Json-LD document!");
120                 return false;
121             };
122 
123             var context = graph.getContext();
124 
125             var node = graph.GetNode(0);
126 
127             // Check if required properties are supplied
128             @@ALLOWED_PROPERTY_CHECK@@
129             @@TEMPLATE@@
130             if (node.hasProperty(context.compactIRI("@@PROPERTY_ID@@")))
131             {
132                 request.SendError("Set on property @@PROPERTY_LABEL@@ not allowed!"
                        );
133                 return false;
134             };
135             @@TEMPLATE@@
136 
137             return true;
138         }
139         @@UPDATE_ALLOWED@@
140 
141         @@CREATE_ALLOWED@@
142         private static bool validatePutRequest(Request request, RuntimeInfo info)
143         {
144             var graph = request.getGraph();
145 
146             if (graph == null || graph.Count() == 0)
147             {
148                 request.SendError("Not a valid Json-LD document!");
149                 return false;
150             };
151 
152             var context = graph.getContext();
153 
154             var node = graph.GetNode(0);
155 
```

```
156             // Check if required properties are supplied
157             @@REQUIRED_PROPERTY_CHECK@@
158             @@TEMPLATE@@
159             if (!node.hasProperty(context.compactIRI("@@PROPERTY_ID@@")))
160             {
161                 request.SendError("Property @@PROPERTY_LABEL@@ not supplied!");
162                 return false;
163             };
164             @@TEMPLATE@@
165 
166             return true;
167         }
168         @@CREATE_ALLOWED@@
169 
170         public static void HandleRequest(Request request, RuntimeInfo info)
171         {
172             try
173             {
174                 switch (request.getMethod())
175                 {
176                     case "GET":
177                     @@GET_ALLOWED@@
178                         // List all
179                         if( String.IsNullOrEmpty(request.getID()))
180                         {
181                             var idlist = Handler.@@OBJECT@@.List();
182 
183                             Graph graph = new Graph();
184 
185                             foreach (var obj in idlist)
186                                 graph.AddNode(obj.toRefNode(info));
187 
188                             request.SendGraph(graph);
189                         }
190                         // Single Get
191                         else
192                         {
193                             var obj = Handler.@@OBJECT@@.Get(request.getID());
194                             request.SendGraph(obj.serialize(info));
195                         }
196                         break;
197                     @@ELSE@@
198                         request.SendError("Get on Type @@OBJECT@@ not allowed!");
199                         break;
200                     @@GET_ALLOWED@@
201                     case "POST": // Edit object with given ID and params
202                     @@UPDATE_ALLOWED@@
203                         Handler.@@OBJECT@@.Update(request.getID(), @@OBJECT@@.
                                fromRequest(request));
204                         request.SendSuccess();
205                         break;
206                     @@ELSE@@
```

```
207                         request.SendError("Update on Type @@OBJECT@@ not allowed!")
                                ;
208                         break;
209                     @@UPDATE_ALLOWED@@
210                     case "PUT": // Create new object
211                     @@CREATE_ALLOWED@@
212                         if(validatePutRequest(request, info))
213                         {
214                             Handler.@@OBJECT@@.Create(@@OBJECT@@.fromRequest(
                                request));
215                             request.SendSuccess();
216                         };
217                         break;
218                     @@ELSE@@
219                         request.SendError("Create on Type @@OBJECT@@ not allowed!")
                                ;
220                         break;
221                     @@CREATE_ALLOWED@@
222                     case "DELETE":
223                     @@DELETE_ALLOWED@@
224                         Handler.@@OBJECT@@.Delete(request.getID());
225                         request.SendSuccess();
226                         break;
227                     @@ELSE@@
228                         request.SendError("Delete on Type @@OBJECT@@ not allowed!")
                                ;
229                         break;
230                     @@DELETE_ALLOWED@@
231                     default:
232                         request.SendError("Method not supported.");
233                         break;
234                 }
235             }
236             catch( Exception e )
237             {
238                 request.SendError(e.Message);
239             };
240         }
241     }
242 }
```

B Handler.Template.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using SPIDER;

namespace @@PROJECT@@.Handler
{
    public class @@OBJECT@@
    {
        // Return list of all available objects
        public static List<Object.@@OBJECT@@> List()
        {
            List<Object.@@OBJECT@@> idList = new List<Object.@@OBJECT@@>();

            // Insert code here
            throw new NotImplementedException("Not implemented!");

            return idList;
        }

        // Return object of given ID
        public static Object.@@OBJECT@@ Get(String ID)
        {
            Object.@@OBJECT@@ obj = new Object.@@OBJECT@@();

            // Insert code here
            throw new NotImplementedException("Not implemented!");

            return obj;
        }

        // Update object with ID the data from the given object
        public static void Update(String ID, Object.@@OBJECT@@ obj)
        {
            // Insert code here
            throw new NotImplementedException("Not implemented!");

        }

        // Delete object with ID
        public static void Delete(String ID)
        {
            // Insert code here
            throw new NotImplementedException("Not implemented!");
        }

        // Create object from given data
        public static void Create(Object.@@OBJECT@@ obj)
```

```
51          {
52              // Insert code here
53              throw new NotImplementedException("Not implemented!");
54          }
55      }
56  }
```

C Server.Template.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using SPIDER;

namespace @@PROJECT@@.Internal
{
    public class ServerWrapper
    {
        RuntimeInfo runtimeInfo;
        Server server;
        String wrapperName;

        public void Run()
        {
            while(true)
            {
                var request = server.WaitForRequest();

                HandleRequest(request);
            }
        }

        private void HandleRequest(Request request)
        {
            switch(request.getObject())
            {
                @@HANDLER_CASES@@
            ##    @@TEMPLATE@@
            ##    case "@@OBJECT_LABEL@@":
            ##        Object.@@OBJECT_LABEL@@.HandleRequest(request, runtimeInfo);
            ##        break;
            ##    @@TEMPLATE@@
                default:
                    request.SendError("Object not in Schema!");
                    break;
            };
        }

        public ServerWrapper(String address, String host, String name, int port)
        {
            wrapperName = name;

            runtimeInfo = new RuntimeInfo()
            {
                HostAddress = host
            };

```

```
51              server = libspider.CreateServer(address, port);
52          }
53      }
54  }
```

D Generierte Klassendefinition für Part

Listing 20: Von SPIDER-Core generierte JSON-LD Klassendefinition für die Klasse Part.

```
{
    "@context" :
    {
        "aras" : "http://spider.thomas-psota.de:8080/Schema/aras/",
        "rdf" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        "rdfs" : "http://www.w3.org/2000/01/rdf-schema#",
        "schema" : "http://schema.org/",
        "spider" : "https://www.thomas-psota.de/spider/schema/Spider.jsonld"
    },
    "@graph" :
    [
        {
            "@id" : "aras:Part",
            "@type" : "rdfs:Class",
            "rdfs:comment" : "Part from PDM System.",
            "rdfs:label" : "Part",
            "spider:create" : "true",
            "spider:delete" : "true",
            "spider:read" : "true",
            "spider:update" : "true"
        },
        {
            "@id" : "aras:PartNumber",
            "@type" : "rdf:Property",
            "rdfs:comment" : "Part number of a Part.",
            "rdfs:label" : "PartNumber",
            "schema:domainIncludes" :
            {
                "@id" : "aras:Part"
            },
            "schema:rangeIncludes" :
            {
                "@id" : "schema:Text"
            },
            "spider:get" : "true",
            "spider:required" : "true",
            "spider:set" : "true",
            "spider:unique_id" : "true"
        },
        {
            "@container" : "@list",
            "@id" : "aras:BOM",
            "@type" : "rdf:Property",
            "rdfs:comment" : "Bill of Material of Part.",
            "rdfs:label" : "BOM",
            "schema:domainIncludes" :
            {
                "@id" : "aras:Part"
```

```
            },
            "schema:rangeIncludes" :
            {
                "@id" : "aras:Part"
            },
            "spider:get" : "true",
            "spider:required" : "false",
            "spider:set" : "false",
            "spider:unique_id" : "false"
        },
        {
            "@id" : "aras:Name",
            "@type" : "rdf:Property",
            "rdfs:comment" : "Name of a Part.",
            "rdfs:label" : "Name",
            "schema:domainIncludes" :
            {
                "@id" : "aras:Part"
            },
            "schema:rangeIncludes" :
            {
                "@id" : "schema:Text"
            },
            "spider:get" : "true",
            "spider:required" : "false",
            "spider:set" : "true",
            "spider:unique_id" : "false"
        },
        {
            "@id" : "aras:Type",
            "@type" : "rdf:Property",
            "rdfs:comment" : "Type of the part (Assembly, Part).",
            "rdfs:label" : "Type",
            "schema:domainIncludes" :
            {
                "@id" : "aras:Part"
            },
            "schema:rangeIncludes" :
            {
                "@id" : "schema:Text"
            },
            "spider:get" : "true",
            "spider:required" : "false",
            "spider:set" : "false",
            "spider:unique_id" : "false"
        }
    ]
}
```

E Ausgefüllter Handler für Commits

```
1  public class Commit
2  {
3      // Return list of all available objects
4      public static List<Object.Commit> List()
5      {
6          List<Object.Commit> idList = new List<Object.Commit>();
7
8          foreach (var commit in Program.repo.Commits) {
9              idList.Add(new Object.Commit(commit.Sha));
10         };
11
12         return idList;
13     }
14
15     // Return object of given ID
16     public static Object.Commit Get(String ID)
17     {
18         var repoCommit
19             = Program.repo.Commits.Where(c => c.Sha == ID).First();
20
21         Object.Commit commit = new Object.Commit();
22
23         commit.SHA = repoCommit.Sha;
24         commit.Author = repoCommit.Author.Email;
25         commit.Message = repoCommit.Message;
26
27         foreach (var parent in repoCommit.Parents) {
28             commit.Parents.Add(new Object.Commit(parent.Sha));
29         };
30
31         return commit;
32     }
33 }
```

F Messwerte für Get-Zugriff

```
Results for http://git.thomas-psota.de:2810/Commit/
---------------------------------
time_pretransfer=0.001513
time_starttransfer=5.190083
time_total=5.240865
---------------------------------
time_pretransfer=0.000725
time_starttransfer=5.097831
time_total=5.151416
---------------------------------
time_pretransfer=0.000748
time_starttransfer=5.315000
time_total=5.360716
---------------------------------
time_pretransfer=0.001122
time_starttransfer=5.179038
time_total=5.239819
---------------------------------
time_pretransfer=0.000685
time_starttransfer=5.301800
time_total=5.349153
---------------------------------
time_pretransfer=0.000677
time_starttransfer=5.095711
time_total=5.149137
---------------------------------
time_pretransfer=0.000689
time_starttransfer=5.071256
time_total=5.134206
---------------------------------
time_pretransfer=0.002269
time_starttransfer=5.275366
time_total=5.327851
---------------------------------
time_pretransfer=0.000898
time_starttransfer=5.086011
time_total=5.144492
---------------------------------
time_pretransfer=0.000738
time_starttransfer=5.057580
time_total=5.107001
---------------------------------
time_pretransfer=0.000757
time_starttransfer=5.134758
time_total=5.201529
---------------------------------

Results for: Results for http://git.thomas-psota.de:2810/Commit/$ID
---------------------------------
time_pretransfer=0.000915
```

```
time_starttransfer=0.455328
time_total=0.455385
---------------------------------
time_pretransfer=0.000701
time_starttransfer=0.458867
time_total=0.458926
---------------------------------
time_pretransfer=0.001018
time_starttransfer=0.486677
time_total=0.487047
---------------------------------
time_pretransfer=0.001142
time_starttransfer=0.452930
time_total=0.453103
---------------------------------
time_pretransfer=0.000838
time_starttransfer=0.477828
time_total=0.477995
---------------------------------
time_pretransfer=0.001064
time_starttransfer=0.506261
time_total=0.506400
---------------------------------
time_pretransfer=0.001033
time_starttransfer=0.479398
time_total=0.479458
---------------------------------
time_pretransfer=0.000778
time_starttransfer=0.449542
time_total=0.449696
---------------------------------
time_pretransfer=0.000708
time_starttransfer=0.528854
time_total=0.529019
---------------------------------
time_pretransfer=0.000841
time_starttransfer=0.508640
time_total=0.508736
---------------------------------
```