



Masterarbeit

Ein System zur partiellen Synchronisation von Wissensbasen für dezentrale soziale Netzwerke

von Jens Grundmann

9. Oktober 2015

Hochschule für Technik und Wirtschaft Berlin

Fachbereich Wirtschaftswissenschaften II

Studiengang Angewandte Informatik

Erstgutachter/in Prof. Vorname Name

Zweitgutachter/in Vorname Name



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

Inhaltsverzeichnis

1	Einleitung	3
2	Grundlagen	4
2.1	Funktionale und nicht funktionale Anforderungen	4
2.2	Shark Framework	5
2.2.1	Context Space	5
2.2.2	Knowledge Base	6
2.2.3	Knowledge Port	6
2.2.4	SyncKB	7
2.3	Datenstrukturen zur Darstellung von Beziehungen	9
2.3.1	Verkettete Listen	10
2.3.2	Bäume	11
2.3.3	Graphen	11
2.3.4	Entity Relationship Modell	12
2.4	Aufbau von Social Media Formaten	12
2.4.1	Chat	12
2.4.2	Forum	13
2.4.3	Dateisysteme und Versionsverwaltung Software	14
2.5	Tags zur Beschreibung von Inhalten	15
3	Konzeption	17
3.1	Serialisierung	17
3.2	Unterscheidung in Beschreibung und Synchronisation	20
3.3	Ein Nachschlagewerk für Datenbereiche	20
3.4	Darstellung eines Datenbereiches über einen Deskriptor	20
3.4.1	Analyse der Datenstrukturen	20
3.4.2	Beschreibung eines Datenbereiches durch einen Deskriptoren	22
3.4.3	Gleichheit von Deskriptoren	23
3.4.4	Darstellung und Bedeutung von Beziehungen	23
3.4.5	Ein Schema für Deskriptoren	26
3.4.6	Extraktion von Daten	27
3.5	Synchronisation	27
3.5.1	Mängel der aktuellen SyncKB	27
3.5.2	Abstraktion von der SyncKB	28
3.5.3	Eliminierung des Piggyback Algorithmus	29
3.5.4	Ein synchronisierbares Schema	30
3.5.5	Ein künstliches Interesse	30
3.5.6	Pull und Pull Request	31
3.5.7	Eine weitere Abstraktionsebene	32
3.6	Lernen von Deskriptoren	33
4	Implementierung	34
4.1	Deskriptor und Schema	34
4.1.1	Die Klassen ContextSpaceDescriptor und DescriptorSchema	34
4.1.2	Serialisierung des Schemas	36

4.1.3	Algorithmen für die Extraktion von Daten	37
4.2	Synchronisation von durch Deskriptoren beschriebenen Datenbereichen	38
4.2.1	Die abstrakte Klasse AbstractSyncKP	38
4.2.2	Ein abstrakte Klasse als Basis	41
4.2.3	Eine Standardimplementierung	43
4.3	Ein Knowledge Port zum Lernen von Deskriptoren	44
5	Tests	45
5.1	Unit Test und Testabdeckung	45
5.2	Codequalität	45
5.3	Test der Anforderungen	45
6	Fazit	47
6.1	Mängel und Verbesserungsmöglichkeiten	47
6.2	Schlusswort	48
7	Quellenverzeichnis	49
8	Abbildungsverzeichnis	51
A	CD-ROM zur Arbeit	52
B	Eigenständigkeitserklärung	53

1 Einleitung

Soziale Netzwerke erfreuen sich immer mehr Beliebtheit in den letzten Jahren, sei es Facebook oder schlicht das Forum zum Online Game, das man gerade spielt. Der Mensch möchte sich austauschen. Allerdings spätestens seit den Enthüllungen Edward Snowdens gegen Ende Mai 2013 stellt sich hier die Frage der Sicherheit der persönlichen Daten. Die meisten sozialen Netzwerke basieren auf einer Client-Server Architektur. Dies bedeutet, dass alle Daten auf einem entfernten Server gespeichert sind. In Bezug auf den Datenschutz muss sich der Nutzer somit auf den Betreiber verlassen, wobei die Server in anderen Ländern stehen können und somit andere Gesetze gelten. Besonders brisant ist dies aufgrund eines Urteils des Europäischen Gerichtshofes, laut dem die Vereinigten Staaten kein sicheres Land für den Datenschutz aller Bürger Europas sind. [1]

Dies ist Motivation für ein Umdenken. Anstatt die Daten an zentraler Stelle zu speichern verbleiben sie auf den lokalen Systemen des Benutzers. Das Client-Server Modell wird durch ein Peer to Peer Modell ersetzt. Daten werden nur mit den Personen ausgetauscht, für die sie bestimmt sind. Hier wird eine Methode benötigt, welche die Daten der lokalen Systeme mit einander synchronisiert.

Ziel dieser Arbeit ist es eine Softwarekomponente zu entwickeln, die eine partielle Synchronisation von Wissensbasen ermöglicht. Eine Wissensbasis ist dabei nichts anderes als eine Menge an Daten, die in einer bestimmten Struktur vorliegen bzw. durch eine abstrakte Darstellung beschreibbar sind. Mittels dieser abstrakte Darstellung soll die Implementierung von Chats, Foren bis hin zu Source Code Management Systemen auf einer Peer to Peer Basis vereinfacht werden. Als Grundlage dient hierzu das Shark Framework [20] von Prof. Dr. Thomas Schwotzer und die darin enthaltene SyncKB Klassensammlung. Sie ermöglicht bereits eine Synchronisation aller Daten in einer Wissensbasis. Diese soll nun dahingehend ausgebaut werden, dass man Teile der Wissensbasis beschreiben kann und die Synchronisation nur so beschriebenen Daten erfassen.

Zuerst soll eine Möglichkeit gefunden werden, wie ein Teilbereich der Wissensbasis beschrieben werden kann. Diese Beschreibungen müssen dann zwischen den einzelnen Peers kommunizierbar und auf den lokalen Systemen der Peers persistierbar sein. Die durch diese Beschreibungen extrahierten Daten werden synchronisiert, wobei ein Peer auf diese Aktion reagieren kann um sie beispielsweise in einer grafischen Oberfläche auszugeben. Hierbei ist darauf zu achten, dass die Beschreibung so abstrakt gewählt wird, dass sie auf möglichst viele Fälle, wie die erwähnte Möglichkeit der Implementierung von Chats oder Foren, anwendbar ist. Als Beweis für die Funktionalität der Softwarekomponente werden Unit-Test geschrieben.

Die Arbeit erläutert zuerst die Grundlagen, wie zum Beispiel das Shark Framework [20], auf denen die Implementierung beruhen wird. Danach soll das Konzept der Softwarekomponente erstellt werden. Dieses diskutiert verschiedene Möglichkeiten, um die im letzten Absatz beschriebenen Anforderungen umzusetzen. Im Anschluss kommt die Beschreibung der eigentlichen Implementierung. Nachfolgend werden die Tests und Methoden zur Qualitätssicherung gezeigt und erklärt. Abschließen tut ein Fazit, welches einen Ausblick auf mögliche Verbesserungen und Erweiterungen gibt.

2 Grundlagen

Das folgende Kapitel widmet sich den Grundlagen auf denen die Arbeit beruht. Neben den funktionalen und nicht funktionale Anforderungen der zu entwickelnden Softwarekomponente werden Frameworks und Modelle besprochen, die zur Entwicklung der Komponente und Umsetzung der Anforderungen genutzt werden können.

2.1 Funktionale und nicht funktionale Anforderungen

Im Folgenden werden die funktionalen und nicht funktionalen Anforderungen besprochen. Diese beschreiben welche Features die Softwarekomponente bereitstellen soll, sowie wichtige Aspekte der Qualitätssicherung.

Funktionale Anforderungen

- **Beschreibbarkeit:** Es ist möglich einen Raum von Daten zu beschreiben und diesen von einem anderen Raum von Daten abzugrenzen.
- **Beziehungen:** Beziehungen zwischen Räumen können definiert werden. So soll beispielsweise der Raum Java-Chat ein Kind des Programmiersprachen-Chat Raumes sein können.
- **Persistenz:** Es besteht die Möglichkeit Beschreibung der Räume von Daten persistent zu speichern. Die gespeicherten Räume bleiben somit erhalten und können zu späterem Zeitpunkt neu geladen werden.
- **Synchronisation:** Die von den Räume beschriebenen Daten und ihre Abhängigkeiten sollen mit Peers in einem Peer to Peer Netzwerk zu synchronisieren sein. Ziel ist es, dass die Räume nach der Synchronisation identisch vom Aufbau und Inhalt sind.
- **Änderbarkeit:** Es muss möglich sein eine Beschreibung eines Raumes zu ändern inklusive seiner Abhängigkeiten. Der Raum selbst soll dabei bestehen bleiben. Das heißt, er muss trotz Änderung mit den gleichen Algorithmen auffindbar sein.
- **Änderung kommunizieren:** Änderungen müssen kommuniziert werden können, sodass sich andere Peers synchronisieren.

Nicht funktionale Anforderungen

- **Build-Management:** Die Softwarekomponente ist mittels eines zu bestimmenden Build Tools so eingerichtet, dass das Aufsetzen der Entwicklungsumgebung für andere Entwickler schnell und einfach zu erledigen ist. Mögliche Synergien des gewählten Tools mit anderen Systemen zur Softwareentwicklung und Qualitätssicherung, zum Beispiel Jenkins [16], sind wünschenswert.
- **Testbarkeit:** Die Softwarekomponente ist modular so aufgebaut, dass Modultest durchgeführt werden können.
- **Modultest:** Es existieren bereits eine Reihe von Modultests, welche die grundlegende Funktionalität der Softwarekomponente sicherstellen.
- **Wartbarkeit:** Der Code der Software soll wartbar sein. Dies bedeutet der Quellcode muss verständlich geschrieben sein und Fehler können relativ schnell entdeckt werden.

2.2 Shark Framework

In diesem Abschnitt wird auf die grundlegenden Features des Shark Framework [20] eingegangen, die für die Arbeit benötigt werden. Das gesamte Framework wird nicht erklärt. Weiterführende Informationen sind im Developer Guide [5] zu finden.

2.2.1 Context Space

Die wichtigste Grundlage ist der Context Space, welcher hier vereinfacht Kontext genannt werden soll. Dabei handelt es sich um eine Datenstruktur. Abbildung 1 zeigt ein vereinfachtes Modell dieser Struktur.

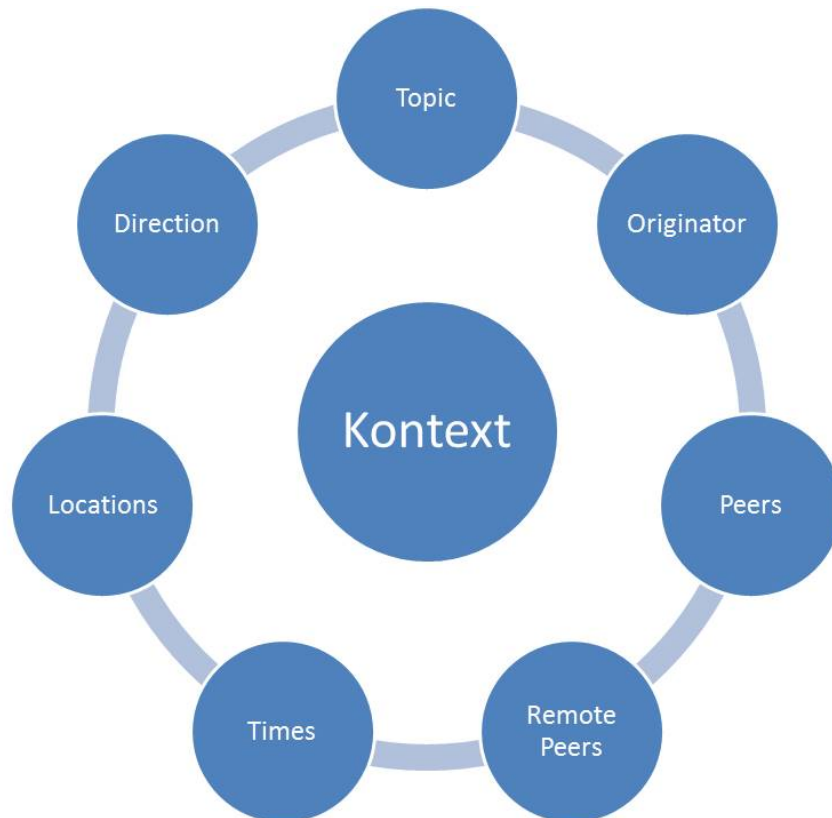


Abbildung 1: Shark Context Space Modell

Die einzelnen Elemente haben dabei folgende Bedeutung:

- **Topic:** Dies ist das Thema eines Kontextes. Es ist eine Beschreibung was der Kontext bedeutet.
- **Originator:** Der Autor des beschriebenen Kontextes. Dies ist entweder der Ersteller sein oder der Autor der beiliegenden Informationen. Bei einem wissenschaftlichen Artikel wäre dies der Autor des besagten Artikels.
- **Peers:** Die Peers Dimension ist unterschiedlich interpretierbar. Zum einen kann es der Ersteller des Kontextes sein, was nicht dem Autor der beiliegenden Informationen entsprechen muss, zum anderen kann es der Besitzer des Kontextes sein, wobei Besitzer hier je nach beiliegendem Fall anders interpretierbar ist. Im Allgemeinen kommt die Interpretation dieser Dimension auf den vorliegenden Fall an.

- **Remote Peers:** Dies sind die Peers an welche der Kontext gesendet werden soll.
- **Times:** Eine Zeitinformation, die je nach vorliegendem Fall anders interpretiert werden kann. Diese Dimension bietet die Möglichkeit ein Zeitintervall anzugeben.
- **Locations:** Gibt einen Ort an. Auch hier ist je nach vorliegendem Fall eine andere Interpretation möglich.
- **Direction:** Die Richtung der Kommunikation des Kontextes. Es ist möglich Daten nur zu empfangen, sie nur zu senden, beides oder keine der genannten Aktionen durchzuführen.

Zu beachten ist, wenn von einem Kontext gesprochen wird, handelt es sich hierbei um eine Beschreibung einer Menge von Kontextpunkten. Diese Punkte haben den gleichen Aufbau wie in Abbildung 1 gezeigt. Der wesentliche Unterschied ist, dass ein Kontext eine Vielzahl von Inhalten, ausgenommen der Direction, besitzen kann. Währenddessen besitzt der Kontextpunkt nur je genau einen Inhalt. Beispielfähig beschreibt der Kontextpunkt nur das Thema Java, während der Kontext die Themen Java und C beinhalten kann. Die Dimensionen selbst werden als Kontextkoordinaten bezeichnet. Der Kontextpunkt vereint Koordinaten und Informationen. Der Kontext wiederum ist eine Möglichkeit, eine Menge an Kontextpunkten aus einer zugrundeliegenden Wissensbasis zu extrahieren.

2.2.2 Knowledge Base

Die Knowledge Base, zu deutsch Wissensbasis, ist eine Sammlung von Wissen. Wissen ist dabei eine Menge an Informationen, die anhand von Kontextkoordinaten beschrieben sind. Die Vereinigung von Kontextkoordinaten und Informationen bilden einen Kontextpunkt.

Die Wissensbasis bietet die Möglichkeit Peers und Themen zu speichern sowie die Themen in einer Taxonomie einzuordnen. Der für die zu entwickelnde Softwarekomponente wichtige Punkt ist aber die Möglichkeit genannte Kontextpunkt in ihr zu speichern und anhand eines Kontextes zu extrahieren. Die Wissensbasis selbst kann hierbei ähnlich einer Datenbank angesehen werden. Ziel ist es, Teilbereiche, sprich eine bestimmte Menge an Daten, mit anderen Wissensbasen zu synchronisieren. Zu diesem Zweck gibt es bereits eine Implementierung, SyncKB genannt, auf der aufgebaut wird.

Zusätzlich zu den genannten Eigenschaften gibt es die Möglichkeit Properties an der Wissensbasis zu speichern. Dies sind Name-Wert-Paare, wobei sowohl Name als auch Wert eine Zeichenkette ist.

2.2.3 Knowledge Port

Das wichtigste Objekt zur Kommunikation im Peer to Peer Netzwerk vom Shark Framework ist der KnowledgePort. Hierbei handelt es sich um eine abstrakte Klasse, welche die Implementierung von *doExpose(SharkCS interest, KEPConnection kepConnection)* und *doInsert(Knowledge knowledge, KEPConnection kepConnection)* erfordert.

Die doExpose-Methode erhält ein Interesse in der Form eines SharkCS. Ist ein Interesse versandt worden, so wird sie als erstes aufgerufen. Ihre generelle Aufgabe besteht darin zu ermitteln, ob das Interesse, das erhalten wurde, für den KnowledgePort von Relevanz ist.

Die doInsert-Methode hingegen erhält ein Knowledge Objekt, welches echte Daten enthält. Es wird in der Regel gegen Ende der Kommunikation aufgerufen und soll die Daten in die Wissensbasis einfügen.

Beide Methoden erhalten ein KEPCConnection Objekt, mit dem sie Informationen über den Sender bekommen können. Des Weiteren werden mithilfe dieses Objektes Daten an die doExpose und doInsert-Methode anderer Peer gesendet. Dies erlaubt einen mehrfachen Datenaustausch. Ein Beispiel hierzu ist der Synchronisationsprozess der SyncKB, der im nachfolgenden Abschnitt beschrieben wird und in Abbildung 2 skizziert ist.

Die Aussagen entsprechen hier dem Standardfall, wonach der Knowledge Port entworfen wurde. Die tatsächliche Implementierung und damit der Umgang mit dem SharkCS Objekt und dem Knowledge Objekt kann von jedem Entwickler selbst bestimmt werden. Einzig vorgeschrieben ist, dass zuerst ein Interesse versandt wird und Daten in der Form eines Knowledge Objektes gesendet werden.

2.2.4 SyncKB

Leider gibt es, abgesehen der JavaDoc Dokumentation, keine genaue Beschreibung der generellen Funktionsweise der SyncKB. Daher soll hier darauf eingegangen werden. Weitere Informationen können dem Quellcode selbst entnommen werden, der im Github vom SharkFW zu finden ist. [13]

Die SyncKB hält den Hauptteil ihrer Logik im SyncKP. Die eigentliche Synchronisation findet während des Kommunikationsprozesses statt und ist keine Eigenschaft der Wissensbasis selbst. Die einzelnen Elemente der SyncKB basieren hauptsächlich auf dem Entwurfsmuster Wrapper. So wurden den einzelnen Element zwei zusätzliche Informationen mitgegeben:

- **Version:** Die aktuelle Version des Elements. Ein neu erstelltes Element startet bei eins und bei jeder Änderung wird die Version um eins erhöht.
- **Zeitstempel:** Ein Zeitstempel der angibt, wann die letzte Änderung an dem Element vorgenommen wurde.

Wie erwähnt befindet sich der Großteil der Logik im SyncKP. Abbildung 2 zeigt die ablaufende Kommunikation zwischen zwei Peers, die hier vereinfacht Alice und Bob genannt werden sollen.

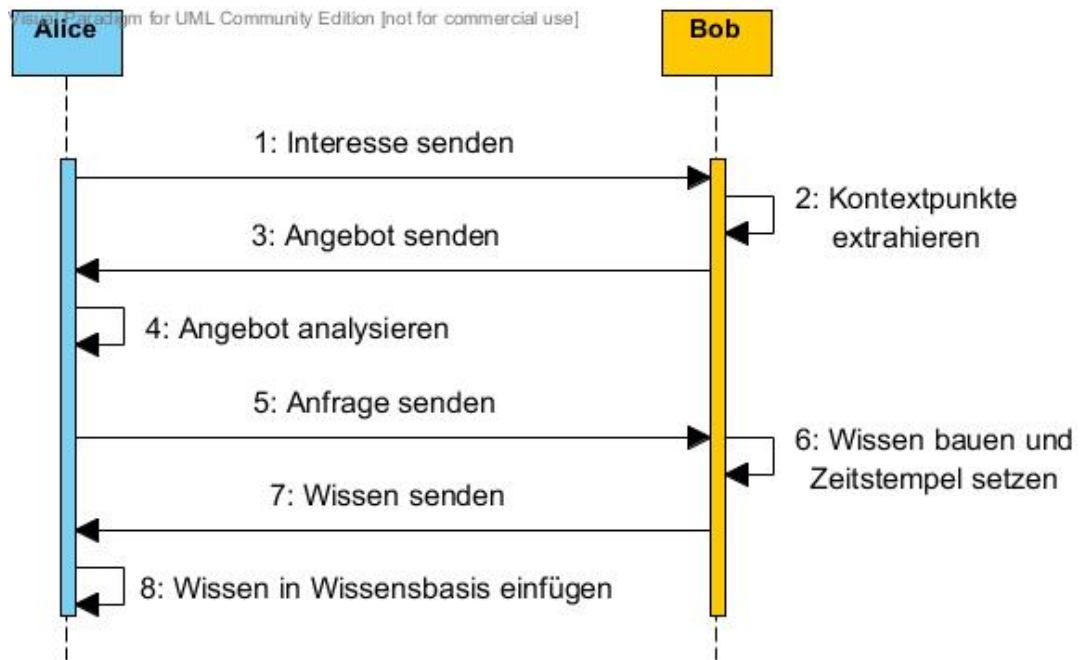


Abbildung 2: Kommunikation der SyncKB

1. **Interesse senden:** Der Prozess beginnt damit, dass Alice ihr Interesse zur Synchronisation verkündet. Das hierbei gesendete Interesse ist ein künstliches Interesse. Es wird vom SyncKP intern verwendet und dient lediglich als Datenhalter für diesen.
2. **Kontextpunkte extrahieren:** Nachdem das Interesse an der Synchronisation Bob erreicht hat stellt dieser ein Angebot für Alice zusammen. Im linken Teil von Abbildung 3 ist der Algorithmus dazu skizziert. Hierbei wird ausgenutzt, dass an jedem SyncContextPoint ein Zeitstempel der letzten Änderung gespeichert ist. Dieser wird mit dem Zeitpunkt des letzten Treffens mit dem Peer, das die Synchronisation anfordert, verglichen. Dazu ist an der Wissensbasis, per Property, eine Liste von Name-Wert-Paare gespeichert, welche den Zeitpunkt des letzte Treffen mit einem Peer enthält. Genauer gesagt ist ein Peer jeweils einem Zeitstempel zugeordnet. Alle Kontextpunkte, deren letzte Änderung neuer ist als das letzte Treffe mit einem spezifischen Peer, hier Alice, werden angeboten.
3. **Angebot senden:** Die extrahierten Kontextpunkte werden per Property am künstlichen Interesse gespeichert und Alice als Angebot zugesandt. Zu beachten ist dabei, dass nur die Daten eines Kontextpunktes gesendet werden. Eventuelle Informationen, die diesem zugeordnet sind, werden nicht versandt.
4. **Angebot analysieren:** Alice analysiert das Angebot, welches sie von Bob erhalten hat. Der Algorithmus ist ähnlich dem späteren Einfügen der Kontextpunkte und im rechten Teil von Abbildung 3 skizziert. Alice wird alle Kontextpunkte von Bob anfragen, die sie nicht besitzt oder die bei Bob eine höhere Versionsnummer haben als bei ihr selbst.
5. **Anfrage senden:** Abermals erfolgt ein Versand der Kontextpunkte per Property am künstlichen Interesse. Diesmal von Alice zu Bob.

6. **Wissen bauen und Zeitstempel setzen:** Nun extrahiert Bobs die angefragten Kontextpunkte aus seiner Wissensbasis und speichert sie in einem Knowledge Objekt. Zusätzlich werden alle Themen und Peers, anhand des beim Erstellen des SyncKP übergebenen FragmentationParameter, dem Knowledge Objekt übergeben. Schließlich wird per Zeitstempel vermerkt, dass eine Kommunikation stattgefunden hat. Dies entspricht dem besprochenen letzten Treffen aus Punkt 2.
7. **Wissen senden:** Das Knowledge Objekt wird nun an Alice gesendet.
8. **Wissen in Wissensbasis einfügen:** Alice überprüft die Kontextpunkte anschließend noch einmal anhand des im rechten Teil von Abbildung 3 skizzierten Algorithmus und fügt diese dann in ihre Wissensbasis ein.

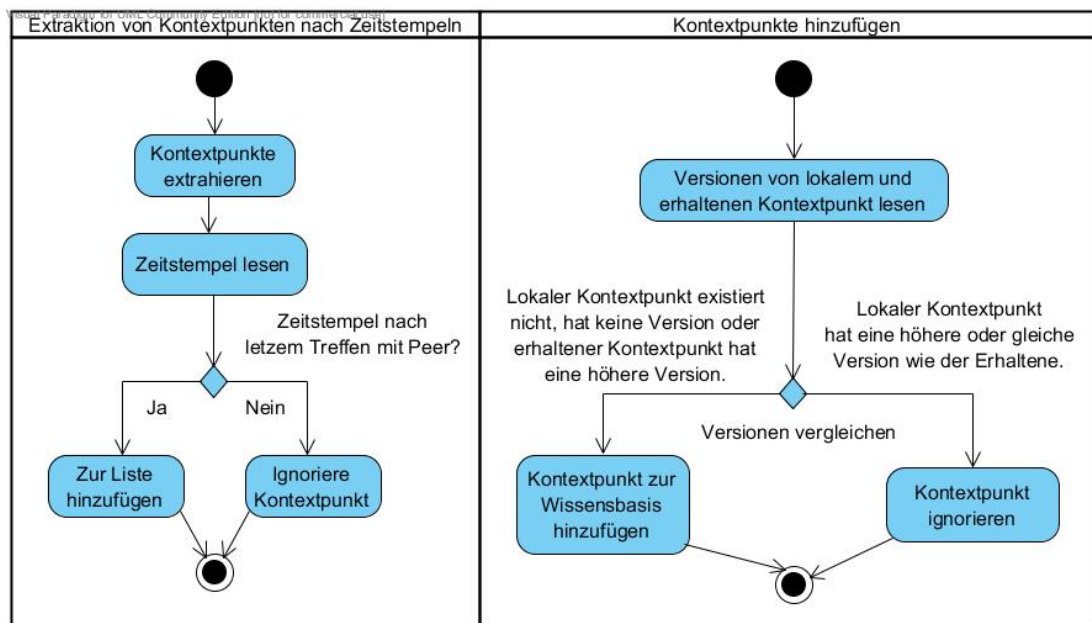


Abbildung 3: Algorithmus zum Extrahieren und Einfügen von Wissen der SyncKB

Das System von Angebot und Anfrage wird verwendet, um das Datenvolumen während der Kommunikation möglichst gering zu halten. Wie erwähnt enthalten Angebot und Anfrage nur die nötigen Informationen zu den Kontextpunkten, wie Koordinaten, Zeitstempel und Version und nicht die mit dem Punkt verknüpften Informationen. Erst gegen Ende des Synchronisationsalgorithmus wird ein Objekt mit den vollständigen Daten erstellt und versandt.

2.3 Datenstrukturen zur Darstellung von Beziehungen

Wie in Abschnitt 2.1 beschrieben, soll es möglich sein Abhängigkeiten zwischen Räumen aufzustellen. Diese Abhängigkeiten stellen eine Beziehung zwischen Daten da. Daher werden in diesem Abschnitt Datenstrukturen besprochen, die eine solche Beziehung abbilden können. Dabei wird darauf eingegangen, wie die Daten und Beziehungen untereinander dargestellt sind. Weitergehende Erklärungen, wie mögliche Operationen, werden nur durch Links zu entsprechender Literatur gegeben.

2.3.1 Verkettete Listen

Verkettete Listen (beschrieben in [4], Kapitel 4) sind Listen, wo jedes Element Referenzen auf weitere Mitglieder der Liste einhält. Hier sollen die Einfach-Verketteten-Listen und die Zweifach-Verketteten-Listen betrachtet werden.

Einfach-Verkettete-Listen

Einfach-Verkettete-Listen besitzen eine Referenz auf ihren Nachfolger. Die Beziehung der einzelnen Elemente ist hierbei, dass jedes Element seinen Nachfolger kennt, nicht aber seinen Vorgänger. Die Beziehungen können also nur vorwärts verfolgt werden, das heißt von einem Element zum Nachfolgenden, nicht aber von einem Element zum Vorherigen. Abbildung 4 skizziert dieses und zeigt mögliche Operation an einer verketteten Liste auf. Weitere Informationen können den Erklärungen unter [18] entnommen werden.

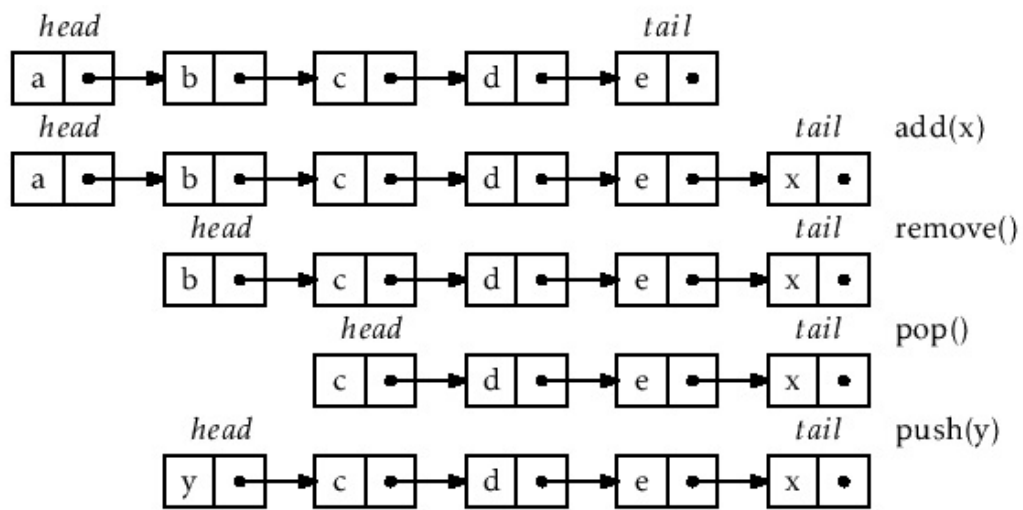


Abbildung 4: Aufbau und Operation von Einfach-Verkettete-Listen. Quelle: [18]

Zweifach-Verkettete-Listen

Zweifach-Verkettete-Listen besitzen, gegenüber Einfach-Verkettete-Listen, eine Referenz auf Vorgänger und Nachfolger. Die Beziehung der einzelnen Elemente ist also, dass jedes Element seinen Vorgänger und Nachfolger kennt. Somit kann die Liste vorwärts, von Element zum nachfolgenden Element, als auch rückwärts, vom Element zum vorhergehenden Element, verfolgt werden. Abbildung 5 skizziert dieses. Weitere Informationen können den Erklärungen unter [19] entnommen werden.

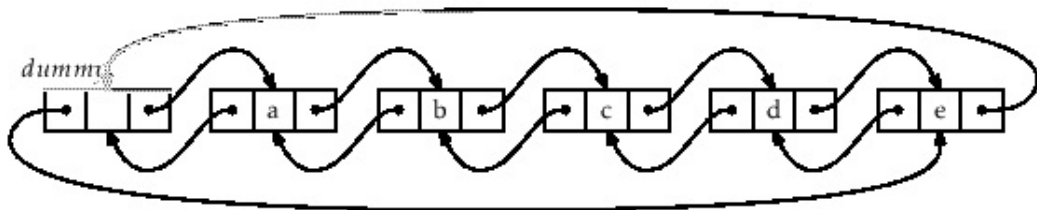


Abbildung 5: Aufbau von Zweifach-Verkettete-Listen. Quelle: [19]

2.3.2 Bäume

Bäume ([4], Kapitel 4) bestehen aus einem Wurzelknoten und weiteren Knoten, die von diesem ausgehen. Hierbei besteht eine Vater-Kind Beziehung zwischen diesen. Die Wurzel hat die besondere Eigenschaft, dass sie keinen Vater besitzt. Knoten, die keine Kinder besitzen, werden Blätter genannt. Ein Element kann hier also mit beliebig vielen Unterelementen, seinen Kindern, in Beziehung stehen. Hingegen kann ein Element von nur einem anderen Element abstammen. Abbildung 6 skizziert ein Beispiel dieser Datenstruktur.

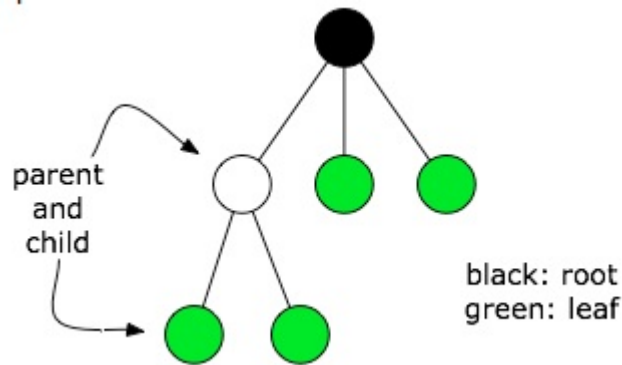


Abbildung 6: Aufbau eines Baums. Quelle: [23]

2.3.3 Graphen

Graphen (siehe [4], Kapitel 6) bilden ein Netz von Daten. Dabei kann von jedem Element eine Beziehung zu einem anderen ausgehen. Gegenüber Bäumen besitzen sie keine Wurzel, welche den Anfang darstellt. Zudem ist die Anzahl an Beziehungen nicht fest gelegt. Ein Element besitzt eine Vielzahl von Beziehungen. Gerichtete Graphen geben dieser Beziehung eine Richtung. Somit kann allerdings nur angegeben werden, von welchem Element man zu welchen gelangt und gegebenenfalls nicht mehr zurück. Dies ist ähnlicher dem Nachfolger Link in einer verketteten Liste und stellt nicht, wie beim Baum, eine Vater Kind Beziehung da. Des Weiteren besteht die Möglichkeit gewichteten Graphen Metadaten, wie die Kosten für einen Übergang von Element A zu Element B, mitzugeben. Dennoch bleibt die Art, wie zwei Elemente im Detail zueinander in Beziehung stehen, unbeschrieben. Abbildung 7 skizziert das Modell eines gerichteten Graphen. Die Pfeile stellen dabei die Übergänge zwischen den Elementen dar. Weitere Erklärungen könne unter [17] gefunden werden.

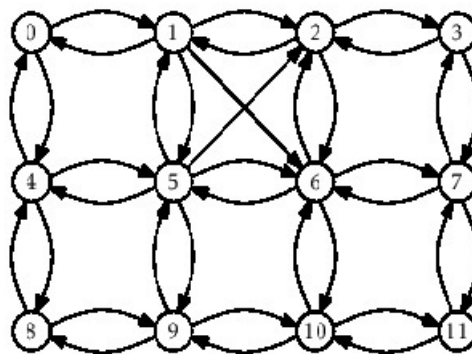


Abbildung 7: Aufbau eines Graphen. Quelle: [17]

2.3.4 Entity Relationship Modell

Das Entity Relationship Modell, beschrieben in [7], ist weniger eine Datenstruktur, als ein Modell zur Beschreibung von Zusammenhängen zwischen Daten, bekannt aus relationalen Datenbanksysteme. Damit beschreibt es aber auch eine Beziehung und kann somit als Grundlage herangezogen werden. Die Beziehungen basieren darauf, dass jedes Element einen eindeutigen Primärschlüssel besitzt. Datensätze, die mit anderen Datensätze, in Beziehung stehen, besitzen einen Fremdschlüssel, welcher identisch zum Primärschlüssel des Datensatzes ist, mit dem sie in Beziehung stehen. Abbildung 8 zeigt eine vereinfachte Skizze dieser Beziehung auf. Anders als die Abbildung erscheinen lässt, sind diese Schlüssel nicht an einen Datentypen gebunden. Auch kann ein Schlüssel mehrere Daten eines Datensatzes umfassen. Die Trennung in Primärschlüssel und Fremdschlüssel erlaubt auch die Art der Beziehung zu interpretieren. Beispielsweise kann dies eine Vater-Kind-Beziehung darstellen. Das Modell ist aber nicht an diese Interpretation gebunden.

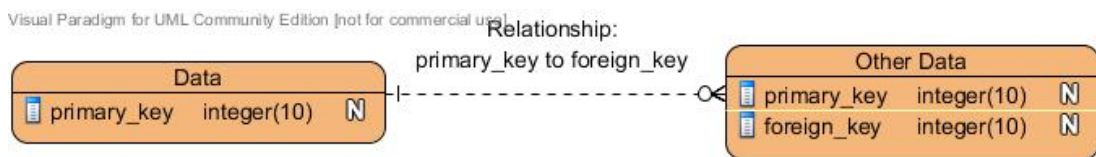


Abbildung 8: Vereinfachte Skizze des Entity Relationship Modell

2.4 Aufbau von Social Media Formaten

In diesem Abschnitt soll der Aufbau einer Reihe von Social Media Formaten beschrieben werden. Social Media Formate sind dabei Anwendungen, die den Kontakt mit anderen Menschen fördern. Die Verknüpfung der einzelnen Informationen und Daten miteinander soll hierbei von besonderem Interesse sein, da sie die Abhängigkeiten aus Abschnitt 2.1 darstellen.

2.4.1 Chat

Chats sind einer der einfachsten Möglichkeiten der Kommunikation. Abbildung 9 zeigt den Aufbau eines Chat im Programm Skype.

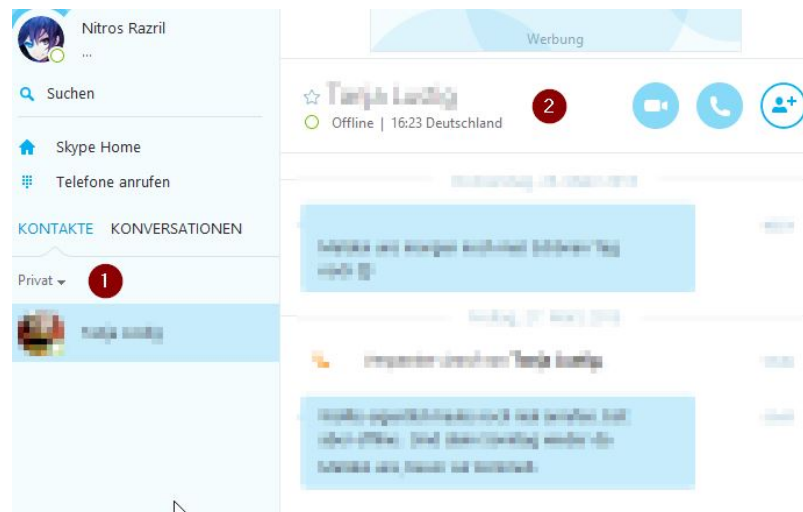


Abbildung 9: Aufbau Chat in Skype. [21]

Die Aufteilung erfolgt hier in erster Ebene (1) in Gruppen. In zweiter Ebene (2) ist der eigentliche Chat. Ein Chat besitzt eine Vielzahl von Daten. Ein Eintrag enthält beispielsweise einen Zeitstempel, Text und Autor der Nachricht. Der Chat selbst erscheint identifizierbar durch einen nicht sichtbaren Identifikator, da Elemente wie Teilnehmer geändert werden können und der Chat trotzdem auffindbar ist. Vereinfacht kann man sagen, dass ein Chat ein einzelner Raum von Informationen ist, der auf eine bestimmte Art durch einen oder mehrere Identifikatoren dargestellt wurde.

2.4.2 Forum

Ein Forum ist gegenüber einem Chat meist komplexer und lässt sich in mehrere Ebenen einteilen. Als Beispiel soll das Burning Board der WoltLab GmbH herangezogen werden [24]. In den folgende Abbildungen 10, 11 und 12 ist der strukturelle Aufbau eines Forums zu sehen.



Abbildung 10: Forumstruktur: Oberste Ebene. Quelle: [24]

Abbildung 10 zeigt die oberste Ebene eines Forums. Man sieht eine Baumstruktur mit einer Wurzel (1). Von diesem Baum gehen Äste (2) aus von welchem wiederum weitere Äste (3) abzweigen können.



Abbildung 11: Forumstruktur: Thread-Sammlung Ebene. Quelle: [24]

Abbildung 11 zeigt die Ebene in der die Threads mit den Inhalten zu finden sind. Man sieht, dass es sich hierbei um einen Knoten (1) in der zuvor erwähnten Baumstruktur handelt. Die einzelnen Threads (2) stellen dabei die Blätter des Baumes dar.

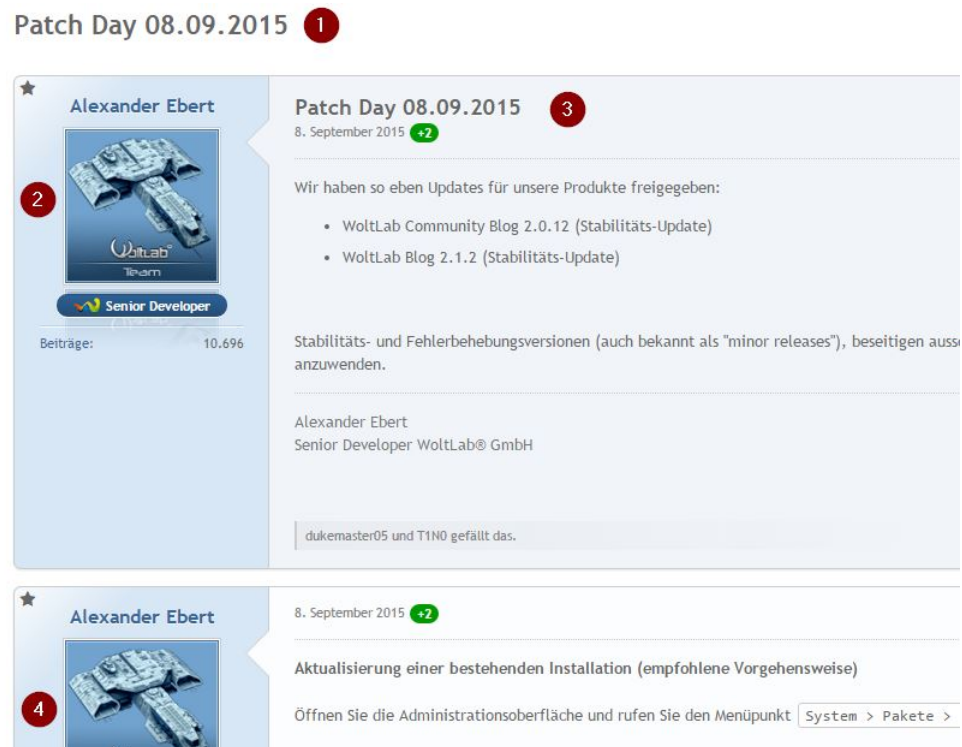


Abbildung 12: Forumstruktur: Thread Ebene. Quelle: [24]

Abbildung 12 zeigt einen Thread in einem Forum. Dieser ist ein Blatt (1) in der Baumstruktur. Ähnlich wie der Chat ist ein Thread ein Raum von Daten. Die einzelnen Einträge (3, 4) besitzen dabei ähnliche Daten zum Chat. Beispiele für diese Daten sind: Zeitstempel, Text und Autor (2).

Trotz des unterschiedlichen Layouts der grafischen Oberfläche besitzen Threads und Chat eine Vielzahl von Gemeinsamkeiten in ihrem Aufbau. Der größte Unterschied ist, dass Foren sich einer Baumstruktur bedienen um die einzelnen Räume von Daten anderen Räumen unterzuordnen. Dies kann als Vater-Kind-Beziehung eines Baumes gesehen werden, wobei hier nur die Blätter Daten enthalten. Andere Knoten dienen lediglich einer Ordnung der Daten und der Eingruppierung dieser in einen definierten Bereich. Jeder Knoten kann aber potenziell Threads enthalten.

2.4.3 Dateisysteme und Versionsverwaltung Software

Es mag auf den ersten Blick nicht so erscheinen, aber auch Dateien auf dem Dateisystem können für soziale Kontakte genutzt werden. Ein einfaches Beispiel hierfür ist die Existenz der vielen Image-Hoster. Auch Facebook und Instagram erlauben den Austausch von Bildern.

In diesem Sinne kann auch Versionsverwaltung Software wie Git oder SVN als eine Art des sozialen Kontaktes gesehen werden. In einem Online Artikel der t3n [3] wird beschrieben, dass man immer mehr auf den webbasierten Filehosting-Dienst für Software-Entwicklungsprojekte GitHub [12] stößt, der auf Git basiert. Mit seiner Vielzahl an Projekten der unterschiedlichsten Programmiersprachen kann GitHub als soziale Plattform für Softwareentwickler gesehen werden.

Aus diesem Grund soll der Austausch von Teilen des Dateisystems über solche Versionsverwaltung Software auch Betrachtung finden. Die eigentliche Versionierung wird dabei vernachlässigt. Abbildung 13 zeigt den allgemeinen Aufbau von Dateisystemen für unterschiedliche Betriebssysteme.

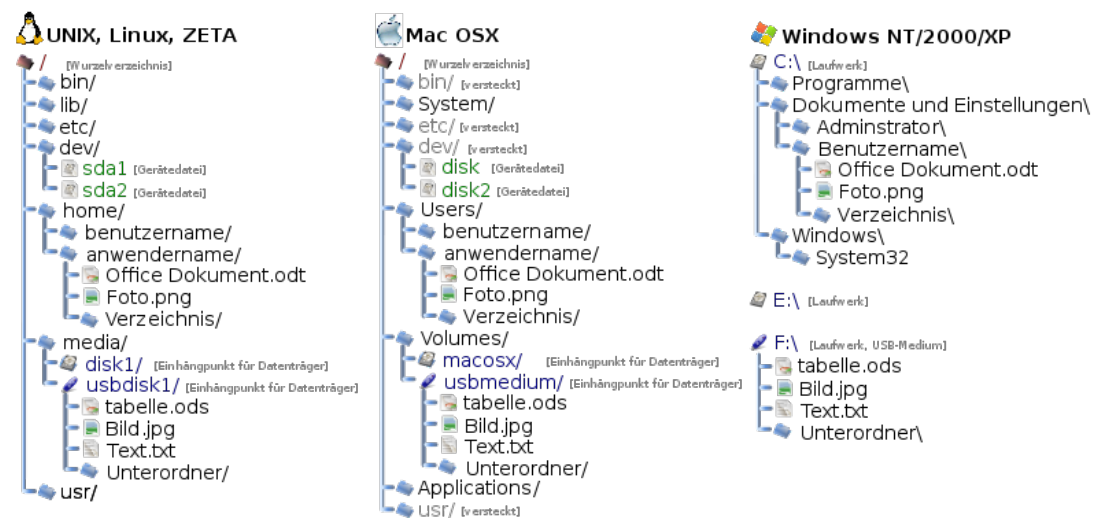


Abbildung 13: Illustration über den Vergleich von Dateisystem-Bäumen. Quelle: [14]

Zu sehen ist, dass identisch zum Forum eine Baumstruktur die Grundlage von Dateisystemen ist. Hier enthalten nur die Blätter die eigentlichen Daten und die anderen Knoten dienen lediglich der Eingruppierung von Daten. Die Blätter selbst können als ein einzelnes Element, bestehend aus Bytes und angereichert mit Metadaten, gesehen werden. Dies unterscheidet sich vom Chat oder Forum, wo ein Blatt ein beschriebener Raum von Daten ist.

2.5 Tags zur Beschreibung von Inhalten

Das sogenannte Tagging von Inhalten ist spätestens seit den Twitter Hashtags bekannt. Es kommt in vielen Formen zum Einsatz. Abbildung 14 zeigt es auf der Frage und Antwort Plattform stackoverflow.

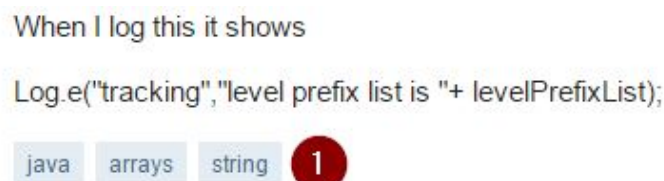


Abbildung 14: Illustration von Tags an einer Frage in stackoverflow. Quelle: [22]

Die Tags (1) dienen dabei der Eingruppierung der Frage. Beim Klick auf eines der Tags gelangt man zu einer Liste von Fragen, die das gleiche Tag tragen.

Was ist der Grund zur Nutzung von Tags? Oded Nov, Mor Naaman und Chen Ye beschreiben es in ihrer Publikation *What Drives Content Tagging: The Case of Photos on Flickr* [9] als eine Möglichkeit, Inhalte mit Metadaten in der Form von Schlüsselwörtern anzureichern. Der Nutzer bedient sich der Tags, damit seine Inhalte besser von anderen gefunden werden. Dies tut er, um seine soziale Präsenz zu erweitern. Scott A. Golder und Bernardo A. Huberman beschreiben Tags in *The Structure of Collaborative Tagging Systems* [6] als eine Möglichkeit Inhalte für zukünftige Navigation, Filterung und Suche zu organisieren. Des Weiteren wird ausgeführt, dass Tags Informationen zu folgenden Aspekten geben können:

- Über was oder wen ist der Inhalt?
- Was für ein Inhalt ist es, ein Buch oder ein Artikel in einer Zeitschrift?
- Wem gehört der Inhalt?
- Einordnung in eine Kategorie
- Bewertung des Inhaltes, wie lustig oder unheimlich.
- Selbstreferenzen, wie *mystuff* oder *mycomments*, um die Relation zu einem selbst darzustellen.
- Organisieren von Aufgaben, beispielsweise Tags wie *toread* oder *jobsearch*.

Im Allgemeinen sind Tags eine Möglichkeit Inhalte zu organisieren. Sie können auch als eine Beschreibung des Inhalts gesehen werden, wobei diese Beschreibung vielen Zwecken dienen kann.

3 Konzeption

In diesem Abschnitt wird die Konzeption der geplanten Softwarekomponente besprochen. Basierend auf den Grundlagen des vorherigen Kapitels wird eine Auswahl getroffen und Techniken zur Implementieren gewählt.

3.1 Serialisierung

Eine der Anforderungen aus dem Abschnitt 2.1 war Persistenz. Es sollte möglich sein eine Beschreibung eines Raumes zu speichern und später wieder abrufen zu können. Zu diesem Zweck macht es Sinn das Feature des Setzen von Properties an einer Wissensbasis zu nutzen. Auf diesem Weg kann je Wissensbasis eine Liste von Beschreibungen gespeichert werden. Insbesondere kann dies als Zuordnung dieser Beschreibungen zur entsprechenden Wissensbasis gesehen werden.

Dazu benötigen wir eine Technik, welche die Serialisierung eines Objektes einer Programmiersprache, in diesem Fall Java, in eine Zeichenkette ermöglicht. Um die Anforderung der Wartbarkeit zu gewährleisten soll hier ein menschlich lesbares Format gewählt werden, dass von den meisten Entwicklern verstanden wird. XML und JSON erfüllen diese Anforderungen. Zum Vergleich sollen die Ergebnisse aus der Fallstudie [8] zu rate gezogen werden. Diese sind in den folgenden Abbildungen dargestellt.

	JSON	XML
Number Of Objects	1000000	1000000
Total Time (ms)	78257.9	4546694.78
Average Time (ms)	0.08	4.55

Abbildung 15: Scenario 1 JSON vs. XML Timing. Quelle: [8]

	Average % User CPU Utilization	Average % System CPU Utilization	Average % Memory Utilization
JSON	86.13	13.08	27.37
XML	54.59	45.41	29.69

Abbildung 16: Scenario 1 JSON vs. XML CPU/Mem. Quelle: [8]

Die Abbildungen 15 und 16 zeigen den ersten Fall der Fallstudie. Hier wurde eine große Menge an Daten über einen Kommunikationskanal gesendet. Es ist klar ersichtlich, dass JSON in den Bereichen Timing und CPU/Memory eine bessere Qualität aufzeigt. Einzig bei der Nutzung der CPU des Nutzers zeigt XML bessere Werte auf.

	JSON	XML
Trial 1 Number Of Objects	20000	20000
Trial 1 Total Time (ms)	2213.15	61333.68
Trial 1 Average Time (ms)	0.11	3.07
Trial 2 Number Of Objects	40000	40000
Trial 2 Total Time (ms)	3127.99	123854.59
Trial 2 Average Time (ms)	0.08	3.10
Trial 3 Number Of Objects	60000	60000
Trial 3 Total Time (ms)	4552.38	185936.27
Trial 3 Average Time (ms)	0.08	3.10
Trial 4 Number Of Objects	80000	80000
Trial 4 Total Time (ms)	6006.72	247639.81
Trial 4 Average Time (ms)	0.08	3.10
Trial 5 Number Of Objects	100000	100000
Trial 5 Total Time (ms)	7497.36	310017.47
Trial 5 Average Time (ms)	0.07	3.10

Abbildung 17: Scenario 2 JSON Vs XML Timing. Quelle: [8]

Trial	Average % User CPU Utilization	Average % System CPU Utilization	Average % Memory Utilization
1	29.07	14.80	67.97
2	83.84	15.84	68.07
3	88.01	11.99	68.06
4	88.65	11.36	68.06
5	88.70	11.30	68.06

Abbildung 18: Scenario 2 JSON CPU/Mem. Quelle: [8]

Trial	Average % User CPU Utilization	Average % System CPU Utilization	Average % Memory Utilization
1	65.80	32.36	68.08
2	67.43	32.57	68.08
3	66.69	33.31	68.08
4	67.24	32.76	68.11
5	66.64	36	68.79

Abbildung 19: Scenario 2 XML CPU/Mem. Quelle: [8]

Die Abbildungen 17, 18 und 19 zeigen den zweiten Fall der Fallstudie. Hier wurden mehrfach hintereinander kleine Datensätze übertragen anstatt alle Datenobjekte in einer Übertragung zu senden. Auch hier zeigt sich, dass JSON im Allgemeinen bessere Werte liefert. Nur bei der Nutzung der CPU des Nutzers zeigt XML, wie im ersten Fall, bessere Werte auf.

Mit den vorliegenden Daten würde die Wahl normalerweise auf JSON fallen. Allerdings handelt es sich bei der zu entwickelnden Softwarekomponente um einen Teil eines Frameworks. Das erfordert eine gesonderte Sichtweise.

- **Abhängigkeiten:** Umgangssprachlich gibt es den Begriff der "Jar-Hölle". Dieser beschreibt, dass Frameworks ihre Abhängigkeiten in der Form von Jar-Archiven mitbringen und dadurch die Möglichkeit besteht diese mehrfach in einem Projekt zu haben. Das erhöht den Speicherbedarf einer Applikation unnötig. Auf der anderen Seite kann es passieren, dass keine Abhängigkeiten mitgebracht werden und von Entwickler eigenständig hinzugefügt werden müssen. Dies erfordert einen ausdrückliche Hinweis in der Dokumentation, wobei nicht sichergestellt werden kann, ob dieser gelesen wird. Zwar kann dies durch das Nutzen eines Tools wie Maven verhindert werden, allerdings nutzt das Shark Framework dieses Tool nicht. Insofern ist interessant, dass Java von Hause aus XML unterstützt mit JAXB. JSON hingegen erfordert das Einbinden eines zusätzlichen Frameworks.
- **Einheitlichkeit:** Das Shark Frameworks benutzt bereits eine XML Repräsentation an vielen Stellen. Ein Mix von XML und JSON bei der Benutzung, kann durchaus verwirrend für Entwickler sein. Besonders im Bezug auf die Weiterentwicklung des Frameworks. Die zur Zeit proprietäre XML Serialisierung sollte einfacher auf das im Java Development Kit enthaltene JAXB umzustellen sein als auf ein externes Framework für JSON, sofern dies bei zukünftigen Refactoring Aktionen geplant ist.
- **XML Schema:** XML bietet die Möglichkeit, den Aufbau der XML Datei über XML-Schema und DDTs zu beschreiben. Dadurch können XML-Parser erkennen, ob ein XML-Dokument valide ist. JSON fehlt diese Möglichkeit. Im Sinne der Wartbarkeit ist dies ein Feature, dass zukünftig an Bedeutung gewinnen könnte. Speziell, da das Shark Framework aus einer Vielzahl komplexer Objekten besteht, die beim Datenaustausch Versand werden.

- **sun.misc.Unsafe:** Wie ein Artikel auf JAXenter [2] beschreibt, plante Oracle ursprünglich für Java 9 die Entfernung von sun.misc.Unsafe. Diese wird dennoch von vielen Frameworks genutzt. Da JAXB Bestandteil des normalen Java Development Kit von Oracle ist, sind hier, gegenüber der Verwendung von Frameworks Dritter, keine Probleme zu erwarten.

Wegen dieser Aspekten, insbesondere dem Punkt Abhängigkeiten, soll hier die Darstellung in XML erfolgen.

3.2 Unterscheidung in Beschreibung und Synchronisation

Per Konzept wird im Rahmen dieser Arbeit in Beschreibung und Synchronisation unterschieden. Die Synchronisation dient zur Darstellung eines Datenbereiches und ist unabhängig vom Algorithmus zum Synchronisieren. Die Synchronisation hingegen baut auf der Beschreibung auf. Sie umfasst ebenfalls den Ablauf der Kommunikation zwischen zwei oder mehr Peers. Dies ermöglicht eine modulare Unterscheidung.

Die Beschreibung dient dem Extrahieren von Daten. Wie mit diesen Daten umgegangen wird ist nicht festgelegt. Währenddessen enthält die Synchronisation die eigentliche Logik zum Ausführen der partiellen Synchronisation.

3.3 Ein Nachschlagewerk für Datenbereiche

Wie in Abschnitt 2.5 beschrieben, eignen sich Tags zum organisieren, filtern und suchen von Inhalten. Die Datenbereiche, die beschrieben werden sollen, können als Inhalte aufgefasst werden. Wenn es also gelingt diese Inhalte mittels eines Kontextes zu beschreiben, so sind sie einfach wiederzufinden. Der Kontext ersetzt dabei das Tag. Gelingt es ebenfalls diese Beschreibungen in Beziehungen untereinander zu setzen, so kann ein Nachschlagewerk für Datenbereiche aufgebaut werden. So könnten sich unter dem Schlagwort Programmiersprachen die Schlagwörter Java und C befinden. Je nach Suchalgorithmus wären so Abhängigkeiten auffindbar und Datenbereiche könnten entsprechend erkannt werden. Die Idee ist hierbei ähnlich der Themen und Peer Taxonomie, welche bereits in einer Wissensbasis existiert. Allerdings ist sie hier auf einen Kontext bezogen.

3.4 Darstellung eines Datenbereiches über einen Deskriptor

In diesem Abschnitt wird besprochen, wie ein Datenbereich dargestellt werden soll. Dazu betrachten wir die Datenstrukturen aus Abschnitt 2.3 und den Aufbau von Social Media Formaten aus Abschnitt 2.4. Interessant dabei ist, inwiefern sie zur Implementierung der Beschreibung des Datenbereiches genutzt werden können, so dass eine Struktur ähnlich der Social Media Formate, erreicht werden kann. Auch inwiefern diese zu XML serialisiert werden können, findet dabei Betrachtung. Zu diesem Zweck wird ein Objekt zur Beschreibung eines Datenbereiches eingeführt: Der Deskriptor.

3.4.1 Analyse der Datenstrukturen

Im folgenden wird auf die Analyse der in Abschnitt 2.3 beschriebenen Datenstrukturen in Bezug auf die in Abschnitt 2.4 genannten Social Media Formate eingegangen.

Verkettete-Listen

Verkettete-Listen erlauben das Darstellen einer Vater-Kind Beziehung. Bei Zweifach-Verketteten-Listen könnte hier vorwärts und rückwärts nach Elementen gesucht werden. Serialisiert wären sie in XML durch eine einfache Liste, mit einer vorgegeben festen Reihenfolge. Das bedeutet, jedes Element müsste im XML in der Reihenfolge erscheinen, wie es in der Liste gespeichert ist.

Verkettete-Listen haben allerdings den Nachteil, dass je nur ein Vorgänger und Nachfolger definiert werden kann. Daher eignen sie sich weniger für die Darstellung der Baumstruktur eines Forums oder eines Teil des Dateisystems.

Bäume

Die meisten der besprochenen Social Media Formate haben von Grund auf eine Baumstruktur. Daher lassen sie sich ohne größere Probleme auch als ein Solcher darstellen. Da XML ebenfalls eine Baumstruktur besitzt, über die Tags als Subtags anderer Tags dargestellt werden können, ist auch die Serialisierung in dieses Format kein Problem.

Problematisch hingegen ist die Tatsache, dass in der Programmiersprache Java, in welcher die Softwarekomponente geschrieben wird, keine vorgefertigte Datenstrukturen für Bäume existiere.

Graphen

Bäume sind prinzipiell eine besondere Art von Graphen. Daher lassen sich die Social Media Formate auch als diese darstellen. Problematisch könnte hierbei sein, die Wege von einem Knoten zu einem anderen zu serialisieren. Dabei kann es sehr einfach zu doppelter Datenhaltung kommen.

Allgemein wären Graphen zu weit gefasst. Kosten für die Wege sind nicht ausschlaggebend für die Beschreibung einer Beziehung. Auch kann angenommen werden, dass man von einem Vater immer zu seinen Kindern kommt und das ein Kind immer seinen Vater kennt. Bäume sind für diesen Aspekt ausreichend, besonders da sie einen Startpunkt, die Wurzel, besitzen.

Entity Relationship Modell

Auch wenn das Entity Relationship Modell eher ein Modell als eine Datenstruktur darstellt, so ist ein Element in diesem trotzdem durch einen Primärschlüssel eindeutig beschrieben. Eine Beziehung zu einem Anderen lässt sich hier einfach durch einen Fremdschlüssel erreichen. Dies ermöglicht die Flexibilität, dass ein Element sehr leicht mehrer Kinder und Väter haben kann, auch als 1:1, 1:N. N:M Beziehung aus Datenbanksystemen bekannt. Viele Elemente der Social Media Formaten sind durch einen Identifikator identifizierbar. Beispielsweise kann ein Thread eines Forums einen numerischen Identifikator besitzen. Dieser kann bei der URL Generierung genutzt werden, um eine eindeutige URL zu erzeugen. Daher sollte es auch möglich sein diesen Identifikator zur Identifikation eines Raumes zu nutzen. Auf übergeordnete Elemente kann mittels des Fremdschlüssel ebenfalls einfach verwiesen werden. Beziehungen sind daher, entsprechen des Namens des Modells, leicht abzubilden. Wenn es sich bei besagtem Identifikator

um einen primitiven Datentypen oder eine Zeichenkette handelt, so ist die Serialisierung nur ein zusätzliches Tag im XML mit ihm als Inhalt.

Problematisch ist, dass die einzelnen Beziehungen programmatisch zusammengesetzt werden müssen. Im Rahmen dieser Arbeit existiert keine relationale Datenbank mit einem Datenbanksystem, welches dies übernehmen könnte.

Auswertung

Da die beschriebenen Social Media Formate bereits eine baumartige Struktur aufweisen ist es sinnvoll auch die Datenstruktur eines Baumes zu nutzen. Um die eigentlichen Vater-Kind Beziehungen darzustellen kann das Prinzip des Entity Relationship Modell aus Datenbanken verwendet werden. Dies ermöglicht einen Identifikator für einen Raum zu definieren, anhand dessen er wieder auffindbar ist. Das Ganze kann als eine Liste serialisiert werden, wobei jeder Identifikator nur einmal vorkommt.

3.4.2 Beschreibung eines Datenbereiches durch einen Deskriptoren

Nach der Analyse aus dem vorhergehenden Abschnitt soll nun festgelegt werden mit welchen Parametern ein Datenbereich beschrieben wird. Abbildung 20 zeigt den Aufbau des beschreibenden Elements, dem Deskriptor.

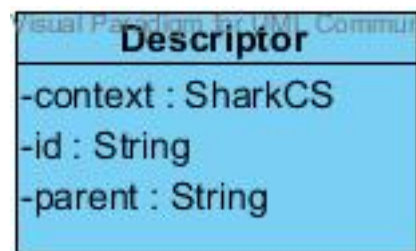


Abbildung 20: Konzeption eines Deskriptor

Dieses Element soll Deskriptor genannt werden. Es besitzt die folgenden Parameter:

- **Kontext (context):** Der Kontext ist der Kern des Deskriptor. Er bestimmt, welche Kontextpunkte durch ihn aus der Wissensbasis extrahiert werden können. Dabei ist es allerdings auch möglich ihm keine Bedeutung zuzuordnen. Dies macht Sinn, wenn der Deskriptor nur zur Beschreibung einer Beziehung benutzt wird, beispielsweise eines Ordners in einem Dateisystem. Dabei ist dann nur interessant, welche Kinder dieser besitzt, nicht aber die Kontextpunkte, die er beschreibt. So ein Deskriptor soll leerer Deskriptor genannt werden.
- **Identifikator (id):** Der Identifikator ist ein Teil des Paares, welche die Vater-Kind Beziehung ermöglicht. Des Weiteren ermöglicht er das Wiederauffinden eines bestimmten Deskriptor aus einer Liste von Deskriptoren.
- **Vater (parent):** Der Vater ist der andere Teil des Paares, welche die Vater-Kind Beziehung ermöglicht. Er erlaubt das Auffinden des Vaters eines Deskriptor. Ebenfalls kann so ein Deskriptor nach seinen Kindern suchen.

Man beachte, dass Identifikator und Vater Zeichenketten sind, statt wie oft übliche numerische Werte. Es ist geplant, dass der jeweilige Entwickler dafür zuständig ist einen Identifikator zu definieren. Dieser sollte daher für Menschen lesbar und interpretierbar sein. Ein Identifikator für einen Chat könnte zum Beispiel einfach Chat genannt werden. Oder es könnte sich um einen Subject Identifier des Topic Maps Modells handeln, dem sich das Shark Framework bedient.

Hierbei sei angemerkt, dass es sich trotzdem um einen technischen Identifikator, der von Maschinen und nicht Menschen verwendet wird, handelt. Die Darstellung als Zeichenkette dient lediglich als Vereinfachung. Somit ist eine größere Auswahl an zu erstellenden Identifikatoren möglich und zeigt Synergien mit dem Topic Maps Modells des Shark Framework auf, da die Subject Identifier ein Array von Zeichenketten sind.

Des Weiteren werden die Väter eines Elements vorerst auf einen Knoten, wie in einem Baum üblich, beschränkt. Dies kann in diesem Modell später, wenn die Notwendigkeit dieser Komplexität besteht, durch das Austauschen des parent Parameters durch eine Liste von Zeichenketten auf mehrere Väter erweitert werden.

3.4.3 Gleichheit von Deskriptoren

Mit der Einführung eines Identifikators für Deskriptoren besitzen diese nun zwei Definitionen von Gleichheit. Die Unterscheidung soll sein, dass ein Deskriptor einem Anderen *gleicht* und ein Deskriptor zu einem Anderen *identisch* ist.

- **Ein Deskriptor *gleicht* einem Anderen:** Deskriptoren sind gleich, wenn ihre Identifikatoren gleich sind, das heißt die Zeichenketten müssen übereinstimmen.
- **Ein Deskriptor ist *identisch* zu einem Anderen:** Deskriptoren sind identisch zueinander, wenn alle ihre Elemente gleich sind, das heißt die Zeichenketten des Identifikator und Vater müssen mit denen, eines anderen Deskriptor übereinstimmen. Des Weiteren muss der Kontext der beiden Deskriptoren nach Regeln des Shark Frameworks identisch sein.

Diese Unterscheidung ist notwendig, um das Wiederfinden eines Deskriptors sicher zu stellen und doppelte Datenhaltung zu verhindern. Anhand des Identifikators kann man Deskriptoren erkennen und nach ihnen suchen. Dennoch muss es möglich sein Unterschiede in den Parametern Vater und Kontext aufzuspüren. Zum Beispiel könnte der Deskriptor von einem anderen Peer geändert worden seien und dann versandt, damit sich alle anderen Peers synchronisieren können. In diesem Fallbeispiel ist es nötig, dass der Deskriptor anhand seines Identifikators gefunden wird und eine Überprüfung stattfindet, ob dieser angepasst werden muss.

Geplant ist die Umsetzung für Gleichheit anhand der equals-Methode, die ein jedes Java-Objekt von Object erbt. Viele Methoden der Collection-API basieren darauf, wie beispielsweise die Aussage, ob eine Liste ein bestimmtes Objekt enthält. Die Aussage ein Deskriptor sei identisch zu einem Anderen, soll anhand einer gesonderten Methode implementiert werden.

3.4.4 Darstellung und Bedeutung von Beziehungen

In diesem Abschnitt wird genauer auf die Darstellung und Bedeutung von Beziehungen zwischen Datenbereichen eingegangen. Die grundlegende Eigenschaft, die einem Deskriptor ermöglichen

eine Beziehung darzustellen, wurde bereits im vorhergehenden Abschnitt erklärt. Das Grundprinzip ist eine Selbstreferenz, die in Abbildung 21 dargestellt ist.

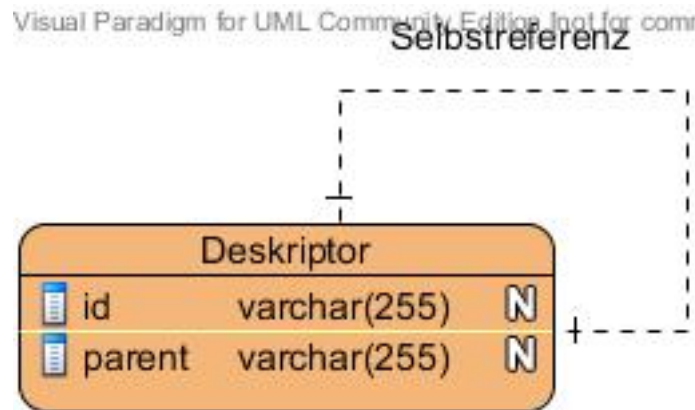


Abbildung 21: Skizze: Selbstreferenz

Die Abbildung zeigt ein Entity Relationship Diagramm als Skizze. Die Zeichenlimitierung an der Zeichenkette kann hier ignoriert werden. Der Datentyp varchar kann verallgemeinert als Zeichenkette interpretiert werden. Die Elemente existieren nur in dieser Form aufgrund der Natur des Diagramms. Der wichtige Aspekt ist die Selbstreferenz. Über den Fremdschlüssel *parent* wird ein Objekt des gleichen Typs referenziert. Dadurch kann einen Baum von beliebiger Tiefe erstellen werden. Ein Vater kann auch beliebig viele Kinder haben, während ein Kind nur einen Vater besitzen kann.

Im Folgenden soll die Anwendung dieses Modells auf die in Abschnitt 2.4 besprochenen Social Media Formate angewandt werden. Es wird gezeigt, wie es konzeptionell angewandt werden kann.

Chat

Die Beziehungen in Chats sind vermutlich am Einfachsten zu beschreiben, da kaum welche existieren. Abbildung 22 zeigt ein Konzept, wie für Beziehungen von Chats.

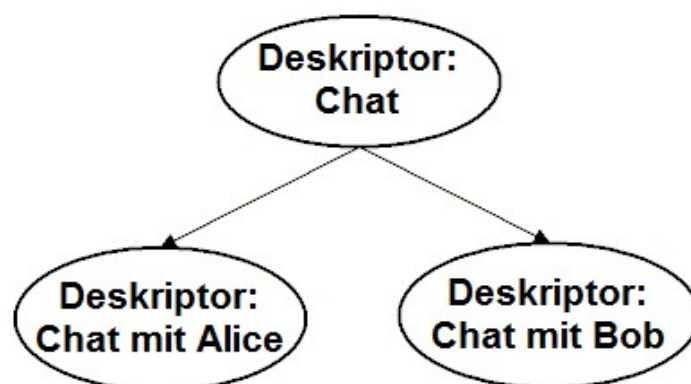


Abbildung 22: Beziehungen von Deskriptoren für Chats

Hier existiert genau eine Vater-Kind Beziehung. Dabei ist die Wurzel des Baumes nur ein Anker, der andere Deskriptoren zusammenfasst. Geht man nach Abbildung 22 so ist es möglich, alle Chats zu finden, indem man alle Kinder des Deskriptor Chats findet. Das Vaterelement soll hierbei keine Informationen oder Kontextpunkte beschreiben. Es dient lediglich der Zuordnung. Der Kontext kann somit als leer angesehen werden. Alle Kindelemente des Deskriptor Chat hingegen beschreiben genau einen Chat. Die zugehörigen Kontextpunkte sind somit die Einträge in dem Chat. Die Beziehung zwischen Deskriptoren wird hier ausschließlich für die Zuordnung zu einer Obergruppe genutzt.

Forum und Dateisysteme

Forum und Dateisysteme gleichen sich in ihrer baumartigen Struktur. Sie können daher ein ähnliches Modell verwenden. Zum einen gibt es die Einordnung in eine Ebene, wie ein Unterforum oder Verzeichnis. Auf der anderen Seite gibt es die Elemente, welche die eigentlichen Daten enthalten.

Abbildung 23 zeigt ein Konzept für Beziehungen, welches die Einordnung von Unterforen und Threads in ein Forum darstellt.

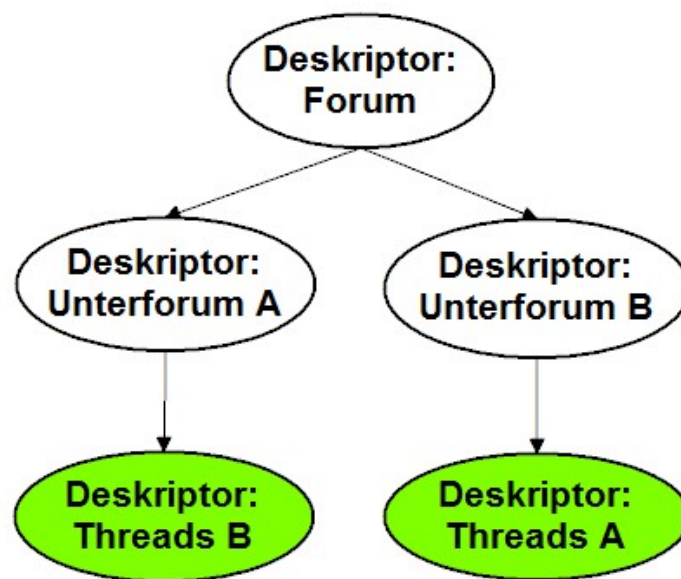


Abbildung 23: Beziehungen von Deskriptoren eines Forum

Abgebildet ist die Baumstruktur. Es wird an einer Wurzel begonnen und die Beziehungen zeigen auf, welche Unterforen von da an existieren. Die Deskriptoren können, wie beim Chat Deskriptor, leer sein und nur zur Darstellung der Beziehung verwendet werden. Zumindest Blätter müssen allerdings eine Menge an Kontextpunkten beschreiben. Per Konzept ist ein Thread eine Menge von Kontextpunkten, wobei jeder Punkt genau einem Post entspricht. Demzufolge können leere Deskriptoren als reine Unterforen angesehen werden. Nicht leere Deskriptoren hingegen schreiben immer Threads. Ein nicht leerer Deskriptor mit einer Beziehung beschreibt Threads und Unterforen.

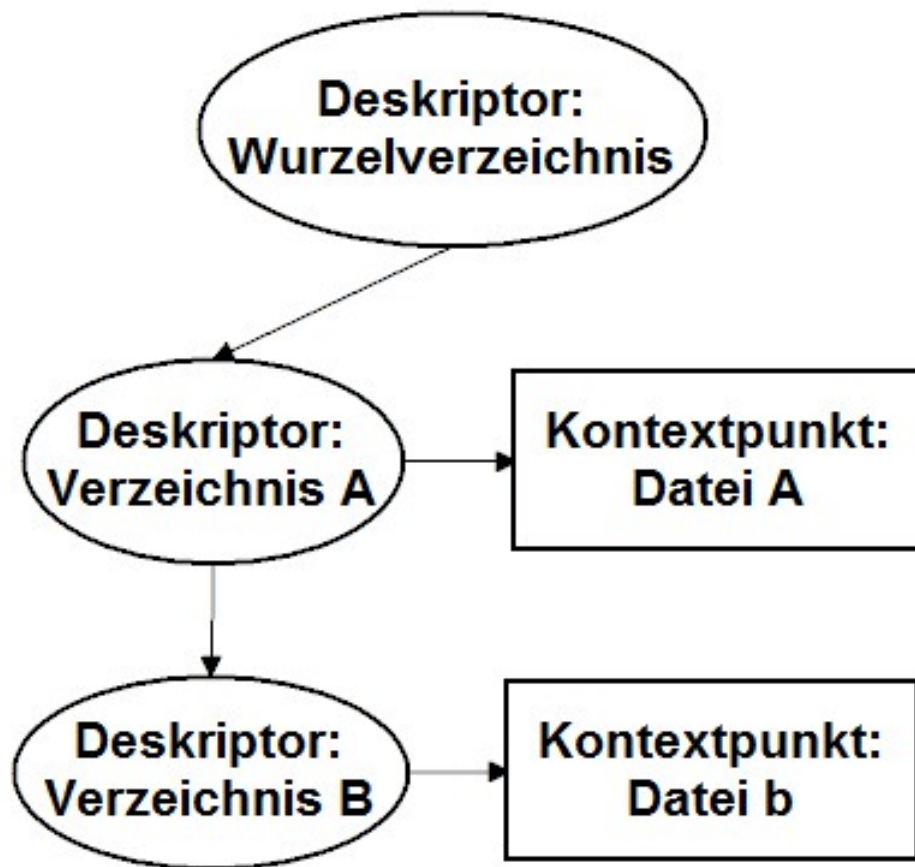


Abbildung 24: Beziehungen von Deskriptoren in einem Dateisystem

Abbildung 24 zeigt die Beziehungen für Verzeichnisse und Dateien. Die Struktur von diesen ist gleich der eines Forums. Der konzeptionelle Unterschied zwischen Dateisystem und Forum ist hier, dass ein Kontextpunkt immer genau eine Datei beschreibt, anstatt dass ein Thread eine Reihe von Kontextpunkten ist.

Man beachte, dass es sich hierbei um ein Konzept handelt. Implementierungen können anders aussehen. Zum Beispiel ist es möglich und eventuell auch sinnvoll, dass eine Datei eine Information an einem Kontextpunkt ist. Somit könnte ein Kontextpunkt mehrere Dateien enthalten. Das Prinzip des Deskriptor ist abstrakt genug um dem Entwickeln hier Freiheit für seine Implementieren zu bieten.

3.4.5 Ein Schema für Deskriptoren

Wie bereits erwähnt gibt es in Java, der Programmiersprache in welcher die zu entwickelnde Softwarekomponente geschrieben werden soll, keine vorgefertigte Datenstruktur für Bäume. Von daher muss eine Klasse erstellt werden, welche dieses ermöglicht. Im folgenden ist beschrieben, welche Aufgaben sie erfüllen soll.

- **Speichern und laden der Deskriptoren:** Das Schema soll die Deskriptoren an der Wissensbasis speichern und aus dieser laden können. Dies soll sowohl für alle gehen als auch für einen bestimmten Identifikator.
- **Vater-Kind Beziehung:** Das Schema stellt die eigentlichen Vater-Kind Beziehungen da. Daher ermöglicht es einem Deskriptor Kinder hinzuzufügen, sowie einen Vater zu setzen. Wird ein neuer Vater gesetzt, so wird der alte überschrieben. Auch muss sichergestellt werden, dass es zu keiner Schleife bei den Bezeichnungen kommt, damit immer ein Baum mit einer Wurzel und Blättern existiert.

3.4.6 Extraktion von Daten

Deskriptoren existieren um Datenbereiche zu beschreiben. Als solches muss die Möglichkeit bestehen Kontextpunkte anhand des Kontext eines Deskriptor zu extrahieren. Dabei sollen die folgenden Möglichkeiten bestehen:

- **Extraktion des Kontext des Deskriptor:** Nur die Kontextpunkte zum Kontext des aktuellen Deskriptor werden extrahiert.
- **Extraktion des Unterbaumes:** Die Kontextpunkte des aktuellen Deskriptor, sowie alle Kontextpunkte seiner Kinder werden extrahiert
- **Extraktion des gesamten Baumes:** Die gesamten Kontextpunkte des Baumes werden extrahiert. Das heißt, es wird zuerst die Wurzel des Baumes gesucht und dann der Unterbaum, inklusive der Wurzel selbst, extrahiert.

3.5 Synchronisation

In diesem Abschnitt wird der Algorithmus zur Synchronisation der Daten konzipiert. Dazu wird der Algorithmus der SyncKB von des Shark Framework mit dem im letzten Abschnitt beschriebenen Deskriptor erweitert.

3.5.1 Mängel der aktuellen SyncKB

Bevor ein Algorithmus entworfen wird sollen die Mängel der aktuellen Synchronisation besprochen werden. Abbildung 25 zeigt nochmal eine Skizze des Algorithmus des SyncKP aus Abschnitt 2.2.4. Für eine Erklärung des sequenziellen Ablauf siehe Abbildung 2.

In der aktuellen Form zeigt der Algorithmus der SyncKB zwei gravierende Mängel auf.

1. **Synchronisation mit allen Peers:** Der Algorithmus nimmt alle Peers der Wissensbasis daher und synchronisiert mit diesen. Für die Fallbeispiele dieser Arbeit soll aber nur mit bestimmten Peers eine Synchronisation stattfinden. So wären dies Beipeilweise in einem Chat die Teilnehmer. In einem Forum alle Mitglieder dieses. Diese Menge ist nicht zwangsweise gleich mit allen Peers in der Wissensbasis.
2. **Synchronisation aller Kontextpunkte:** Ebenfalls synchronisiert der Algorithmus alle Kontextpunkte einer Wissensbasis. Dem Namen dieser Arbeit nach ist allerdings eine partiellen Synchronisation erwünscht.

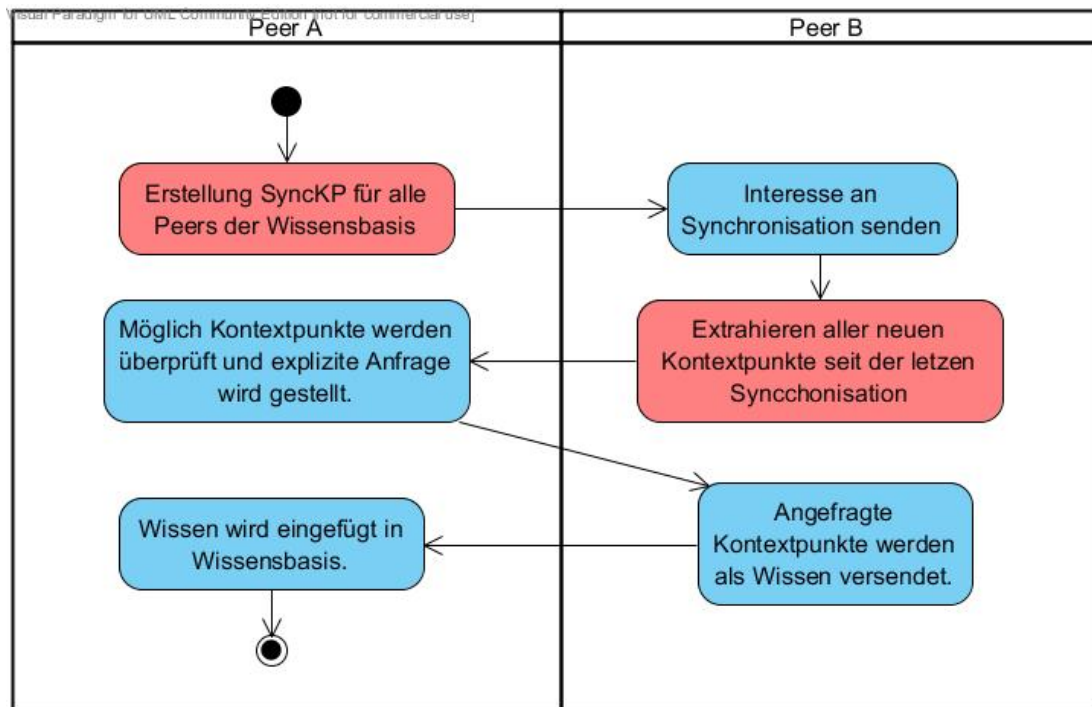


Abbildung 25: Skizze: Synchronisation Algorithmus der aktuellen SyncKB

3.5.2 Abstraktion von der SyncKB

Aufgrund der im letzten Abschnitt beschriebenen Mängel soll eine Abstraktion der SyncKB vorgenommen werden. Genauer soll die Klasse SyncKP abstrahiert werden. Wenn hier von einer Abstraktion gesprochen wird, dann ist von einer Abstraktion im Sinne der objektorientierte Programmierung die Rede. Die Klasse SyncKP soll in eine abstrakte Klasse AbstractSyncKP umgewandelt werden. Der Entwickler, der von dieser erbt, ist dann verpflichtet die abstrakten Methoden zu implementieren. Ziel der Methoden ist das Erreichen folgender Aktionen:

- **Entscheidung ob ein Interesse besteht:** Es sollte im Ermessen des Entwicklers liegen, ob eine Interesse für eine Knowledge Port interessant ist oder nicht. Je nach Fall kann es vorkommen, dass bestimmte Dimensionen des gesendeten Interesse überprüft werden müssen oder nicht. Dem Entwickler soll hier Freiheit geben werden dies selbst zu entscheiden.
- **Finden des Identifikator:** Die bestehende Implementation der SyncKB benötigt einen Identifikator woran es seine Metadaten speichern kann. Dieser Identifikator ist nicht gleich mit dem Identifikator eines Deskriptor. Es handelt sich um ein tatsächliches Tag an dem Properties gespeichert werden können. Im künstlichen Interesse der aktuellen SyncKB ist dies eine Tag in der Topic Dimension, dass schlicht dem halten von Metadaten dient.
- **Zusammenstellen des Angebotes:** Wie in Abschnitt 3.4.6 erläutert gibt es mehrere Möglichkeiten Wissen anhand eines Deskriptor zu extrahieren. Daher ist es sinnvoll dem Entwickler zu überlassen wie genau das Wissen aus der Wissensbasis extrahiert wird.

3.5.3 Eliminierung des Piggyback Algorithmus

Wie in Abbildung 2 aus Abschnitt 2.2.4 zu sehen ist benutzt die aktuelle SyncKB Implementierung einen Piggyback Algorithmus. Sinn dieses ist es den Datenaustausch möglichst gering zu halten. Erste Test mit einer abstrakten Version der SyncKP Klasse haben aber gezeigt, dass dies zu einer langen Laufzeit des Algorithmus führe kann, selbst wenn die Daten nur lokal ausgetauscht werden. Dieser führte sogar in vielen Fällen zu einer `SocketException` mit der Fehlerbeschreibung *Software caused connection abort: socket write error*. Dadurch wurde der Algorithmus abgebrochen ohne das Wissen gesendet wurde und die Synchronisation schlug fehl. Nach einem Post auf stackoverflow [11] kann dies auftreten, wenn der Empfänger nie den Empfang der Daten bestätigt. Da der Fehler nach der Vereinfachung verschwand ist anzunehmen, dass die Komplexität des Algorithmus gekoppelt mit weiteren Algorithmen zur partiellen Synchronisation die Ursache hierfür ist. Aus diesem Grund wurde der Piggyback Algorithmus entfernt.

In Abbildung 26 ist der, verglichen mit Abbildung 2, vereinfachten Algorithmus skizziert.

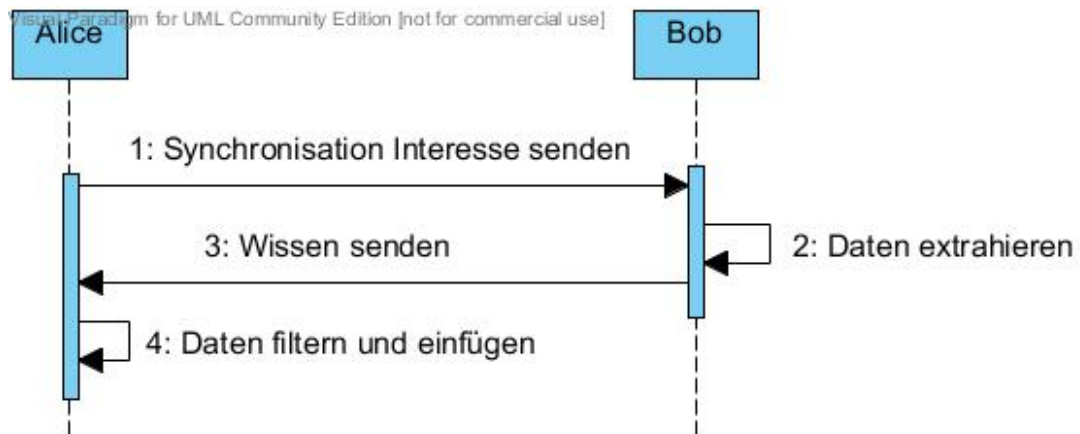


Abbildung 26: Vereinfachter Algorithmus zur Synchronisation

Der Ablauf wurde auf folgende Schritte vereinfacht:

1. **Interesse senden:** Wie im alten Algorithmus versendet Alice zuerst ein Interesse an einer Synchronisation.
2. **Daten extrahieren:** Bob erreicht diese Interesse. Bob wird nun explizit das Wissen extrahieren. Er macht keinen Vorschlag an möglichen Kontextpunkte sondern sendet das Wissen direkt zu. Auch hier werden wieder alle Kontextpunkte und zugehörige Themen und Peer gesendet, die nach dem letzten Treffen mit dem Kommunikationspartner aktualisiert wurden.
3. **Wissen senden:** Das gesamte Wissen wird nun an Alice gesendet. Alice prüft nicht mehr, welche Kontextpunkte sie benötigt und stellt auch keine Anfrage nach bestimmten Kontextpunkten mehr. Die erhält alles, was Bob extrahiert hat.
4. **Daten filtern und einfügen:** Alice überprüft nun, welche Kontextpunkte sie einfügen will. Dies ist identisch zu dem Algorithmus, der im rechten Teil von Abbildung 3 skizziert und unter der Abbildung erklärt ist.

Vorteil dieser Eliminierung ist, dass der Algorithmus leichter zu verstehen ist und der Quellcode somit leichter gewartet werden kann. Nachteil ist, dass potenziell eine große Menge an Wissen versendet werden, der Kommunikationspartner aber nur einen geringen Teil einfügt. Dies ist aber dem Erzeugen eines Fehlers vorzuziehen. Besonders, da in der zu entwickelnden Softwarekomponente nur Teilbereiche von Wissensbasen synchronisiert werden sollen. Daher kann davon ausgegangen werden, dass die Datenmenge generell relativ klein ausfallen. Zusätzlich ist es einem Nutzer leicht verständlich zu machen, dass das Senden von großen Datenmengen eine gewisse Zeit beanspruchen kann. Dies kennt er eventuell von dem Versenden von E-Mails mit großen Anhängen. Eine lange Laufzeit einer Sequenz von Befehlen ist eher selten verständlich für Nutzer.

3.5.4 Ein synchronisierbares Schema

Das Schema ist nicht von sich aus synchronisierbar. Grund hierfür ist, dass es auf einer einfachen Wissensbasis beruht. Für die Synchronisation wird aber eine Wissensbasis benötigt, die als SyncKB implementiert ist. Da diese aber, wie in Abschnitt 2.2.4 erklärt, nur ein Wrapper ist, kann hier per Ableitung eine Kindklasse erstellt werden. Diese nimmt anstatt einer normalen Wissensbasis nur eine SyncKB entgegen und bietet eine Methode an diese per Getter vom Schema zu erhalten. Da bei der Synchronisation oft der Eigentümer der Wissensbasis benötigt wird, soll dieser ebenfalls überprüft werden. Wenn dieser fehlt, so soll bei der Initialisierung ein Fehler auftreten.

Über diesen einfachen Schritt wird ein Schema erzeugt, das immer auf einer SyncKB beruht und dessen Wissensbasis immer einen Eigentümer hat. Die abgeleitete Klasse ist somit ein Vertrag zur Versicherung, dass diese Bedingungen erfüllt sind.

3.5.5 Ein künstliches Interesse

Wie bei der ursprünglichen Implementierung der SyncKB soll auch hier ein künstliches Interesse genutzt werden. Der Grund hierfür ist, dass ein Interesse im Shark Framework keinen Platz für Metadaten bietet.

Die Dimensionen sind nachfolgend beschrieben. Abbildung 27 zeigt eine Skizze für den Aufbau des künstlichen Interesses.

- **Topics:** Die Topics Dimension soll genau ein SemanticTag enthalten. Dieses ist ein Thema, das keiner Beschreibung dient. An diesen sollen nur Properties gesetzt werden, die Metadaten entsprechen. Der Deskriptor selbst soll als XML an ihr gespeichert werden. Somit wird die Topics Dimension zu einer Metadaten Dimension umdefiniert. In dieser Dimension wird sich daher auch immer nur dieses eine Thema finden, welches die Metadaten hält.
- **Peers:** Die Peers Dimension stellt die Person da, die das Interesse an einer Synchronisation versendet. Geplant ist, dass dies immer nur eine einzelne Person ist. Genauer der eigene Eigentümer des Schemas, zu dem der Deskriptor gehört.
- **Remote Peers:** Die Remote Peers Dimension sind die Personen, bei denen eine Synchronisation angefragt wird. Das heißt von diesen Personen wird Wissen empfangen und eventuell in die Wissensbasis eingefügt. Gegenüber anderen Dimensionen können sich in

dieser mehrere Elemente befinden. Das heißt man kann sich mit mehreren anderen Personen synchronisieren. Die aktuellsten Daten werden dabei in die Wissensbasis eingefügt.

- **Direction:** Die Direction Dimension gibt an, ob ein Knowledge Port aktiv ist. Dies kann ähnlich einer Subscription gesehen werden. Ist ein Knowledge Port abonniert, so wird dieser Daten versenden und empfangen. Dies entspricht dem Verhalten der Direction INOUT. Ist ein Knowledge Port nicht abonniert, so wird er weder Daten senden noch sie empfangen. Dies entspricht dem Verhalten der Direction NOTHING.

Die Dimensionen Originator, Times und Locations sind für den Algorithmus zur Synchronisation unerheblich und bleiben daher leer.

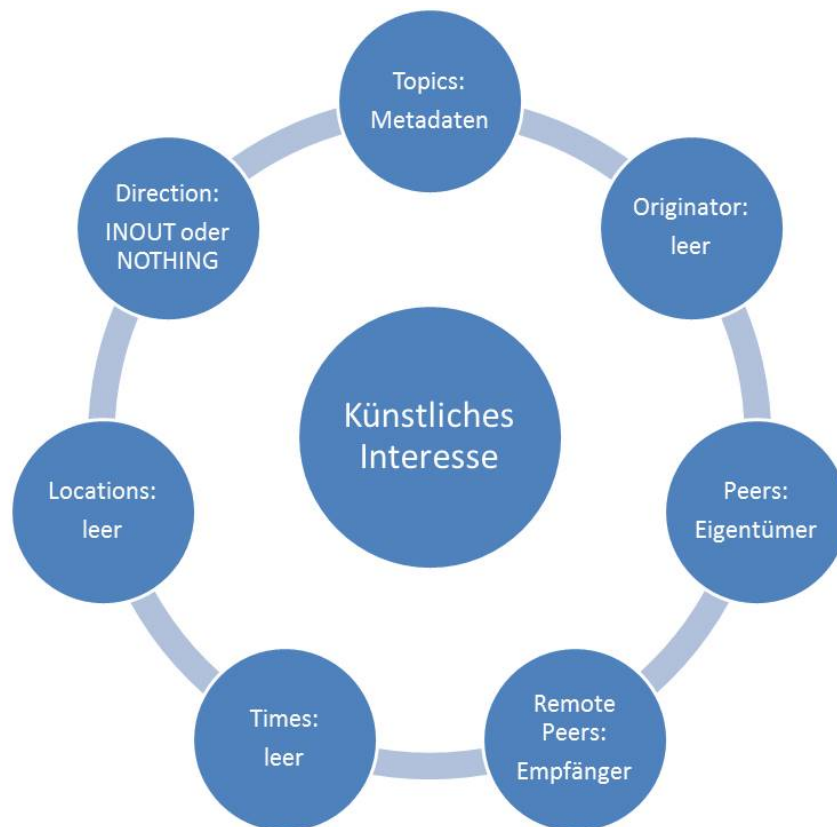


Abbildung 27: Skizze: Künstliches Interesse für partielle Synchronisation

3.5.6 Pull und Pull Request

In dieser Arbeit, sowie in der Implementierung der vorhandenen SyncKB, ist von Synchronisation die Rede. In der Realität ist der beschriebene Algorithmus aber eher einem Pull, bekannt aus der Versionsverwaltung Software Git, gleichzusetzen. Dies bedeutet, dass die Synchronisation einseitig ist. Nur der Peer, welcher das Interesse sendet, bekommt Daten zugesandt. Die Wissensbasis des Peers, der die Daten sendet, ändert sich nicht. Das heißt, dass Peer A alle Daten, die durch einen Deskriptor beschrieben sind, von Peer B erhalten wird. Peer B hingegen wird eventuell nicht alle Daten von Peer A besitzen, die durch den Deskriptor beschrieben sind.

Zu diesem Zweck soll in zwei Aktionen unterschieden werden: Pull und Pull Request.

- **Pull:** Ein Pull entspricht dem in Abbildung 26 skizzierten Algorithmus. Hier werden Daten von einem entfernten Peer geholt und in die eigene Wissensbasis eingefügt. Der Kommunikationspartner erhält keine Daten und somit ändert sich sein Datenbestand nicht.
- **Pull Request:** Ein Pull Request ist nichts anderes als eine andere Person um ein Pull zu bitten. Dies hat zur Folge, dass diese Person alle Daten vom Antragsteller bekommt. Der eigene Datenbestand ändert sich hier nicht. Nur die Wissensbasis der Person, welche die Anfrage erhält, wird aktualisiert.

Eine vollständige Synchronisation wäre somit ein Pull gefolgt von einem Pull Request. Im Rahmen dieses Konzeptes soll dies nicht implementiert werden. Grund hierfür ist, dass zum Zeitpunkt dieser Arbeit keine Möglichkeit besteht zu erkennen, wann die Kommunikation eines Knowledge Port beendet ist. Daher ist nicht klar, wann ein Pull beendet ist und der Pull Request gestartet werden kann. Dieser Aspekt wird somit als Mangel des zugrundeliegenden Algorithmus gesehen und aus zeitlichen Gründen vernachlässigt.

3.5.7 Eine weitere Abstraktionsebene

Neben der Abstraktion des SyncKP ist eine weitere Abstraktionsebene geplant. Das künstliche Interesse sowie die Nutzung der Deskriptoren soll in dieser Ebene. Abbildung 28 zeigt eine Skizze der Abstraktionsebenen, die verwendet werden sollen.

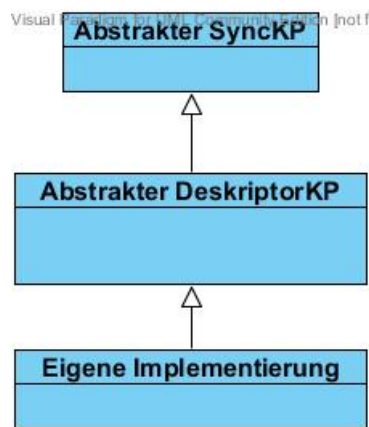


Abbildung 28: Skizze: Abstraktionsebenen

In der ersten Ebene liegt der eigentliche Algorithmus zur Synchronisation. In der zweiten Ebene wird dieser so angepasst, dass er auf Grundlage eines Deskriptor geschieht. Die letzte Ebene wird von Entwickler selbst implementiert. Sie enthält alle Funktionen, die vom Anwendungsfall abhängen.

Geplant ist, dass Deskriptor Ebene folgende Funktionen anbietet bzw. übernimmt:

- **Senden eines Deskriptors:** Sollte der Deskriptor geändert werden, so kann die Änderung an die Kommunikationspartner verteilt werden. Die anderen Peers überschreiben dabei den Deskriptor in ihrem Schema mit dem neuen Inhalt.

- **Senden von geänderten Adressaten:** Die Personen, mit denen eine Synchronisation erfolgen soll, können sich ändern. Die Deskriptor Ebene bietet an Adressaten hinzuzufügen und zu entfernen. Wie mit dieser Änderung umzugehen ist muss allerdings jeder Entwickler in der letzten Ebene selbst bestimmen.
- **Finden des Identifikators:** Es wird das künstliche Interesse genutzt. Das Thema in der Topic Dimension, das die Metadaten hält, soll dabei als Identifikator dienen.
- **Interesse bekunden:** Die Deskriptor Ebene übernimmt das Prüfen des Interesses. Sie entscheidet, ob eine Kommunikation stattfindet. Dies ist der Fall, wenn die Deskriptoren die Gleichheit aufweisen, also ihr Identifikator gleich ist.
- **Subskription:** Je nach Status einer Subskription wird der Knowledge Port Daten versenden und empfangen oder inaktiv sein.

Für die eigene Implementierung durch den Entwickler bleiben somit folgende Tätigkeiten übrig:

- **Senden eines Deskriptors:** Der Entwickler bestimmt, wie genau Daten extrahiert werden. Dies entspricht dem Zusammenstellen des Angebotes der Abstraktion des SyncKP (vergleiche Abschnitt 3.5.2).
- **Auf geänderten Adressaten reagieren:** Es liegt in der Zuständigkeit eines jeden Entwicklers wie auf Änderungen der Adressaten reagiert wird. Eine entsprechende Vorgehensweise muss implementiert werden.

3.6 Lernen von Deskriptoren

Es wurde ein Konzept zum Aufbau von Deskriptoren, sowie einer Synchronisation auf deren Grundlage erstellt. Da Deskriptoren aber nicht immer vordefiniert vorhanden sein müssen, soll auch ein Konzept zum Lernen dieser erstellt werden. Dabei soll sich einer sehr einfachen Methode dient werden.

Um anderen Kommunikationspartner ein Deskriptor beizubringen wird er über einen, von den anderen Algorithmen unabhängigen, Knowledge Port versandt. Dabei nicht der Deskriptor selbst übertragen, sondern sein gesamter Baum. Andere Peers fügen diesen Baum dann in ihr Schema ein. Dabei werden alle existierenden Deskriptoren mit dem gleichen Identifikator überschrieben. Diese kann natürlich unerwünscht sein, weshalb die entsprechende Klasse so gebaut sein soll, dass über Ableitung eine Anpassung des Verhaltens möglich ist.

4 Implementierung

In diesem Kapitel wird die Implementierung besprochen. Es wird gezeigt, wie das Konzept in Java Quellcode implementiert wurde und welche Techniken dazu verwendet wurden. Des Weiteren wird besprochen, wie entworfenen Klassen genutzt werden können.

4.1 Deskriptor und Schema

Im folgenden Abschnitt wird auf die Implementierung des Deskriptor und Schema eingegangen. Es wird erklärt, welche Eigenschaften sie haben und wie sie benutzt werden können.

4.1.1 Die Klassen ContextSpaceDescriptor und DescriptorSchema

Deskriptor und Schema werden in die zwei Klassen ContextSpaceDescriptor und DescriptorSchema aufgeteilt. Die Klasse ContextSpaceDescriptor ist dabei eine reine Beschreibung eines Datenbereiches. Die Klasse DescriptorSchema organisiert die Deskriptoren in einer Baumstruktur und ermöglicht das Speichern und Laden dieser von einer Wissensbasis. Abbildung 29 zeigt das Klassendiagramm beider.



Abbildung 29: Klassendiagramm: Deskriptor und Schema

Konstruktoren und private Methoden wurden hier vernachlässigt. Eine explizite Beschreibung aller Klassen kann der JavaDoc Dokumentation gefunden werden (siehe Anhang A).

Im folgenden soll auf die Verwendung dieser Klassen eingegangen werden.

Deskriptor

Die Klasse ContextSpaceDescriptor hat die aus Abschnitt 3.4.2 beschriebenen drei Parameter Kontext (context), Vater (parent) und Identifikator (id). Für dieses Methoden gibt es jeweils Setter und Getter Methoden. Zu beachten dabei ist, dass der Setter für den Vater auf die Sichtbarkeit protected gesetzt ist. Der Entwickler soll die Vater-Kind Beziehungen über das Schema konfigurieren, welche Fehlerprüfungen durchführt, nicht über die Klasse ContextSpaceDescriptor selbst. Das der Getter für parent die Sichtbarkeit public besitzt dient nur zu Debugging und Logging Zwecken.

Des Weiteren gibt es eine Reihe von Methoden, welche folgende Eigenschaften prüfen:

- **Leere:** Die isEmpty Methode prüft ob ein Deskriptor leer ist. Ein Deskriptor ist leer, wenn der Parameter context null ist.
- **Zwei Deskriptoren sind gleich:** Die von Object überschrieben equals Methode prüft ob zwei Deskriptoren gleich sind. Ein Deskriptor gleicht einem anderen, wenn die Parameter id gleich sind. Mit dieser Prüfung auf Gleichheit werden Features des Collection Frameworks von Java genutzt. Die contains Methode zum Beispiel gibt so wieder ob in einer Collection bereits ein Deskriptor mit dem gleichen Identifikator existiert. Zusätzlich ist auch die hashCode Methode überschrieben. Das ausschlaggebende Element ist auch hier der Parameter id. Demzufolge ist auch der Hash zweier Deskriptoren gleich, wenn ihr Identifikator gleich ist.
- **Zwei Deskriptoren sind identisch:** Die identical Methode prüft ob ein Deskriptor identisch zu einem anderen Deskriptor ist. Dies ist der Fall, wenn Die Parameter id und parent die Gleichheit aufweisen, sowie die context Parameter identisch sind nach den Regeln des Shark Frameworks. Somit können Deskriptoren auf Unterschiede geprüft werden, obwohl sie den gleiche Identifikator besitzen.

Schema

Die Klasse DescriptorSchema ermöglicht sowohl das Speichern, wie auch das Laden von Deskriptoren aus der Wissensbasis. Des Weiteren können die Beziehungen zwischen Deskriptoren konfigurieren werden. Nach außen ist sie die Klasse, welche die Beziehungen enthält, auch wenn die Schlüssel intern an der Klasse ContextSpaceDescriptor beschrieben sind. Sie kann ähnlich einem Datenbankmanagementsystem gesehen werden.

Ein Schema basiert immer auf einer Wissensbasis, aus der die Daten geladen und gespeichert werden. Des weiteren besitzt sie folgende Features:

- **Speichern und Laden von Deskriptoren:** Deskriptoren können an der Wissensbasis, die das Schema verwendet, gespeichert und geladen werden. Hierzu enthält die Klasse die Methoden `getDescriptor`, `getDescriptors`, `saveDescriptor`, `saveDescriptors`, `removeDescriptor` und `saveDescriptors`. Das Löschen des gesamten Schemas ist mittels `clearDescriptors` möglich. Deskriptoren werden immer in einem Set gehalten, das heißt alle Elemente der Liste sind unterschiedlich. Unterschiedlich bezieht sich hierbei auf die Gleichheit, das heißt sie besitzen alle unterschiedliche Identifikatoren. Das Speichern eines Deskriptor mit dem gleichen Identifikator wie ein bereits existierender Deskriptor kommt einem Überschreiben gleich.
- **Konfigurieren von Vater-Kind Beziehungen:** Über die Methoden `addChild`, `addChildren`, `setParent` können Vater-Kind Beziehungen aufgebaut werden. Zu beachten ist, dass ein Deskriptor immer nur einen Vater haben kann. Wird ein neuer gesetzt, so wird der Alte überschrieben. Das Hinzufügen eines Kindes zu einem Deskriptor ist gleich dem Setzen eines Vaters (Parameter: `descriptor`) für den Parameter `child`. Existiert ein Deskriptor nicht im Schema oder würde das Hinzufügen zu einer Schleife führen, also die Baumstruktur verletzen, so erzeugt dies einen Fehler. Ein Vater kann über die `clearParent` Methode gelöscht werden. Über die `getChildren` und `getParent` können Vater und Kinder eines Deskriptor erhalten werden.

Neben den dargestellten Methoden existieren weitere statische Methoden um Eltern und Kinder in einer `java.util.Collection` zu finden. Auch existieren Methoden die Aussagen geben, ob sich im Schema bereits ein gleicher oder identischer Deskriptor befindet. Weiteres kann in der JavaDoc Dokumentation (siehe Anhang A) nachgeschlagen werden.

4.1.2 Serialisierung des Schemas

Wie in Abschnitt 3.1 erklärt soll XML unter Zuhilfenahme des im Java Development Kit enthaltenen JAXB zum Serialisieren des Schemas verwendet werden. Hierzu sind die Klasse `ContextSpaceDescriptor` mit entsprechenden Annotations ausgestattet. Für den Parameter `context` wurde eine entsprechende `XmlAdapter` (siehe [15]) geschrieben, der die Serialisierung einer Shark-CS Klasse unter Zuhilfenahme der im Shark Framework existierenden Klasse `XMLSerializer` für JAXB ermöglicht.

Aufgrund der Annotations wurde auch darauf verzichtet ein Interface zur Klasse `ContextSpaceDescriptor` zu erstellen oder sie abstrakt zu gestalten. Sie wurde als normales Java Objekt entworfen das einfach wie eine Java-Bean genutzt werden kann.

In der JAXB 2.2 Specification, Section 4.2 JAXB Context [10] steht folgendes beschrieben:

”To avoid the overhead involved in creating a `JAXBContext` instance, a JAXB application is encouraged to reuse a `JAXBContext` instance.”

Aus diesem Grund wurde das Factory Pattern für die Erstellung eines JAXBContext benutzt. Abbildung 30 skizziert dieses.

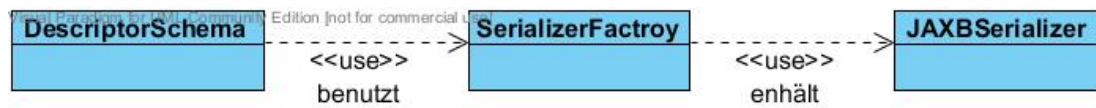


Abbildung 30: Skizze: Factory Pattern für JAXB Serialisierung

Der JAXBContext ist in der Klasse JAXBSerializer enthalten. Für jede Instanz dieser Klasse wird ein neuer JAXBContext angelegt. Um dies zu verhindern wird die Factory Klasse SerializerFactory benutzt. Diese folgt dem Singleton Pattern und enthält genau einen JAXBSerializer der für Deskriptoren konfiguriert ist. Das Schema kann diesen über eine Methode abrufen. Sollten weitere Klassen einen JAXBSerializer für Deskriptoren benötigen, so kann die Instanz über SerializerFactory wiederverwendet werden. Somit wird auch der JAXBContext wiederverwendet und Overhead vermieden.

4.1.3 Algorithmen für die Extraktion von Daten

Wie in Abschnitt 3.4.6 beschrieben gibt es drei Arten der Extraktion von Daten. Als Implementierung wurde hierzu die Klasse DescriptorAlgebra geschrieben. Diese hält nur statische Methoden, welche die entsprechenden drei Extraktionen ausführen.

Extraktion des Kontextes eines Deskriptor

Für den Algorithmus der Extraktion des Kontextes eines Deskriptor wird die vom Shark Framework bereitgestellte Klasse SharkCSAlgebra genutzt. Die Kontextpunkte werden anhand des Kontextes des Deskriptor und einer übergeben Wissensbasis extrahiert.

Extraktion des Unterbaumes

Der Algorithmus für die Extraktion eines Unterbaumes ist im linken Teil von Abbildung 31 dargestellt. Zuerst wird das Wissen des aktuellen Knoten wie bei der Extraktion des Kontext eines Deskriptor extrahiert. Danach werden die Kinder des aktuellen Deskriptor ermittelt. Existieren Kinder, so kommt es zu einem rekursiven Aufruf. Dieser wird bis zu den Blättern des Baumes laufen und das Wissen eines Blattes zurückliefern. Danach wird im Vaterknoten das Wissen des Kindes mit dem Wissen des Vaters vereint (nicht abgebildet) und das Wissen des nächsten Kindes wird extrahiert. Dies geschieht bis zum Anfangsknoten. Schließlich wird das Wissen dieses mit dem Wissen der Kinder vereint. Somit werden alle Kontextpunkte des aktuellen Deskriptor, sowie aller seiner Kinder, gefunden. Dabei wird auch auf Gleichheit dieser geprüft, damit keine Kontextpunkt im Wissen doppelt vorkommt. Die Gleichheitsprüfung erfolgt über die von Object geerbte equals Methode einer jeden Java Klasse.

Extraktion des gesamten Baumes:

Bei der Extraktion des gesamten Baumes wird zuerst die Wurzel gesucht. Das heißt es wird solange die Vaterkette entlang gegangen, bis ein Knoten in dieser keinen Vater mehr besitzt. Im Anschluss erfolgt eine Extraktion des Unterbaumes. Da hier von der Wurzel ausgegangen wird entspricht eine Extraktion des Unterbaumes einer Extraktion des gesamten Baumes. Skizziert ist diesen im linken teil von Abbildung 31.

Die Suche nach der Wurzel in diesem Algorithmus ist auch der Grund, warum im Schema streng eine Baumstruktur genutzt wird mit je einem Vater und keine Schließen auftauchen dürfen. Andernfalls könnte keine Wurzel gefunden werden.

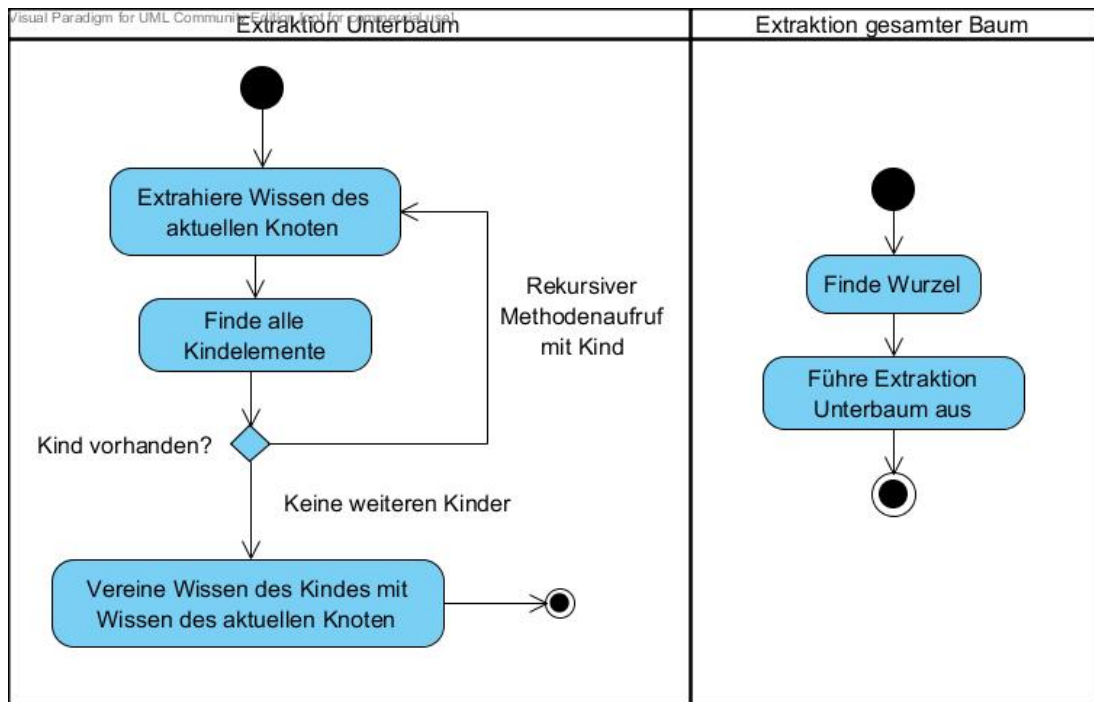


Abbildung 31: Skizze: Algorithmus Extraktion

4.2 Synchronisation von durch Deskriptoren beschriebenen Datenbereichen

Im folgenden Abschnitt werden die Techniken und Algorithmen besprochen, die zur Synchronisation von durch Deskriptoren beschriebenen Datenbereichen genutzt werden.

4.2.1 Die abstrakte Klasse AbstractSyncKP

Die Klasse SyncKP wurde abstrahiert um sie allgemeiner einsetzen zu können. Die erwähnten Eliminierung des Piggyback Algorithmus aus Abschnitt 3.5.3 ist hierbei die größte Änderungen. Dadurch fallen die Phasen Angebot und Anfrage weg. Außerdem hat die neue Klasse AbstractSyncKP einige neue Methoden. Die Wichtigsten sind im Klassendiagramm in Abbildung 32 dargestellt.

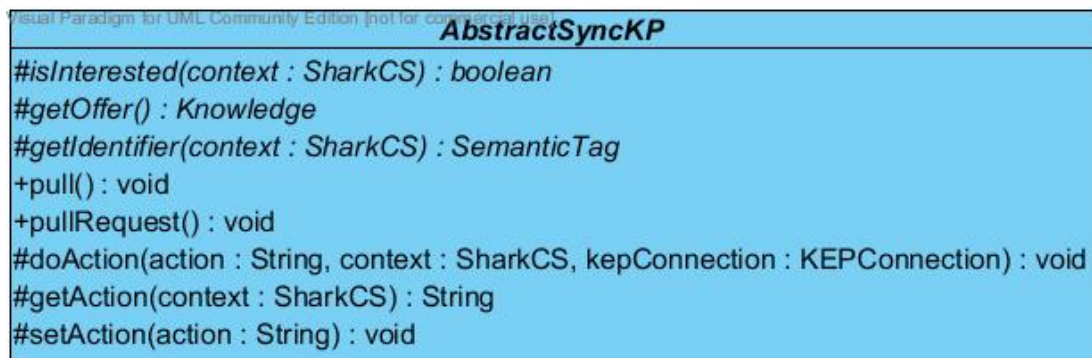


Abbildung 32: Klassendiagramm: AbstractSyncKP

Die abstrakten Methoden besitzen die Sichtbarkeit protected. Sie müssen von jeder Kindklasse implementiert werden. In Abbildung 33 ist hervorgehoben, wo die Methoden während der Synchronisation Verwendung finden werden.

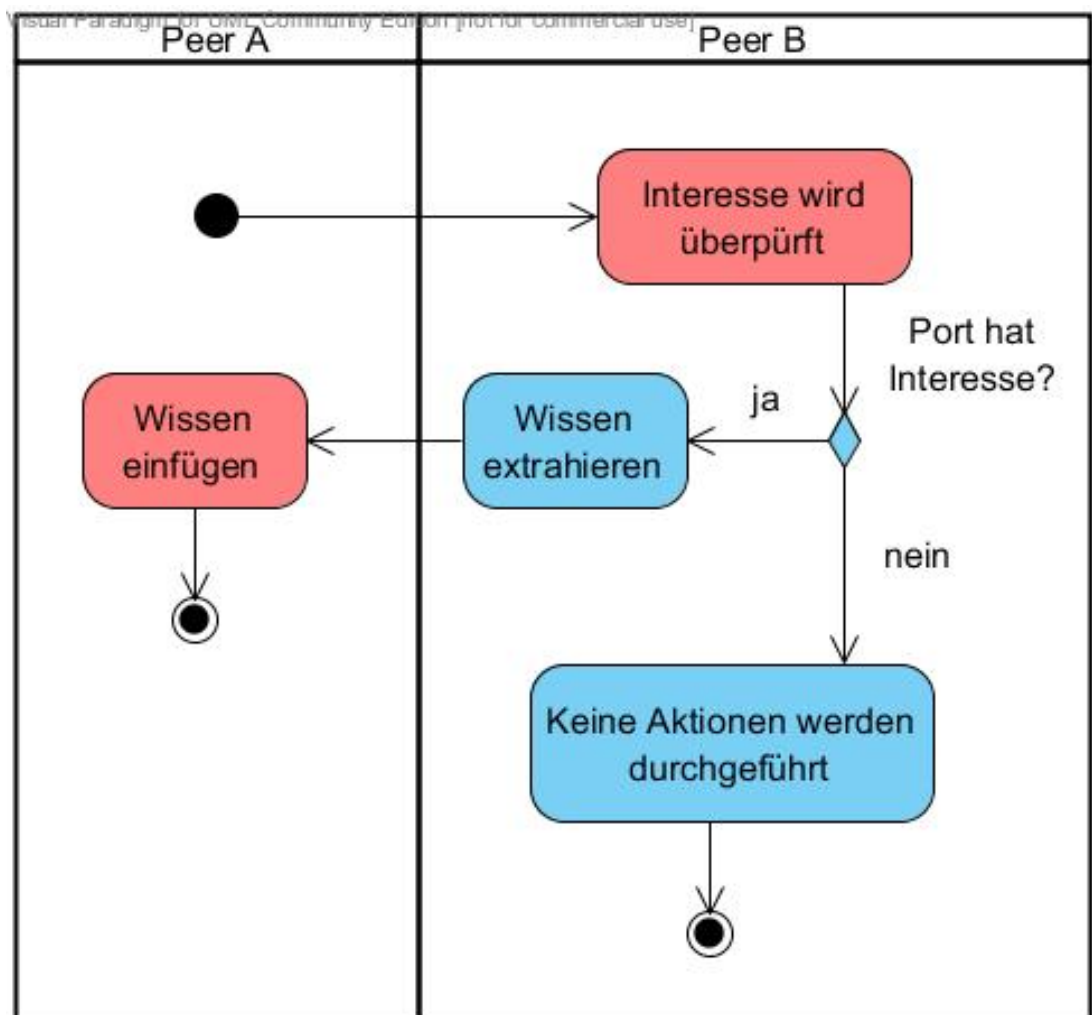


Abbildung 33: Ablauf einer Synchronisation

- **Interesse bekunden:** Zuerst muss herausgefunden werden, ob das erhaltene Interesse überhaupt relevant ist. Die Implementierung hierzu ist jedem Entwickler selbst überlassen. So kann beispielsweise gegen eine Blacklist geprüft werden, da man mit bestimmten Personen gar nicht kommunizieren möchte. Natürlich muss hier auch implementiert werden, ob das erhaltene Interesse überhaupt ein Interesse zur Synchronisation ist. Nur wenn diese Methode true zurückliefert werden weitere Aktionen durchgeführt.
- **Wissen extrahieren:** Die Extraktion des benötigten Wissen kann sich je nach Einzelfall unterscheiden. Dem Entwickler wird hier die Freiheit gelassen, wie er dies implementieren möchte. Das hier erhaltene Wissen wird noch einmal gefiltert und nur Daten, die sich seit dem letzten Treffen mit dem Kommunikationspartner geändert haben, werden versandt.
- **Identifikator finden:** Der Identifikator wird benutzt um Metadaten zu speichern und zu versenden. Dabei ist dieser nicht identisch mit dem Identifikator eines Deskriptor. Vielmehr handelt es um ein SemanticTag, das in einer der Dimensionen des Interesses des Knowledge Port existiert.

Man beachte, dass der Identifikator im eigentlichen Algorithmus keine Funktion hat. Er wird benutzt um zwischen den Aktionen Pull und Pull Request zu unterscheiden, die im Folgenden erklärt sind.

Aktionen Pull und Pull Request

Wie in Abschnitt 3.5.6 erklärt existieren zwei Arten um eine Synchronisation durchzuführen. Pull und Pull Request. Ein Pull ist der in Abbildung 33 dargestellte Ablauf. Ein Pull Request ist eine Bitte an einen Kommunikationspartner diesen Algorithmus zu starten.

Implementiert sind die Aktionen in den Methoden pull und pullRequest. Um zu ermitteln welche Aktion ausgeführt werden soll findet eine Prüfung von Metadaten statt. Diese sind am Identifikator des erhaltenen Interesses als Property gespeichert. Dabei gilt: Ist die Property leer, so entspricht dies der Aktion Pull. Für andere Aktionen, wie dem Pull Request, wird die Property auf Gleichheit mit bestimmten vordefinierten Zeichenketten überprüft.

Die beiden Aktionen werden daher wie folgt gestartet:

- **Pull:** Da die Nichtexistenz einer Aktion einem Pull entspricht wird die Kommunikation ohne weitere Vorbereitungen eingeleitet. Der Kommunikationspartner erkennt, dass keine Aktion definiert wurde und starten den Ablauf eines Pull.
- **Pull Request:** Vor der Kommunikation wird die Aktion Pull Request im zugehörigen Metadaten am Identifikator gesetzt. Nach dem Senden wird dieses Feld wieder entfernt. Der Kommunikationspartner liest die Metadaten und erkennt die Aktion Pull Request. Daraufhin wird er die Methode pull an sich selbst aufrufen, wodurch er selbst ein Pull ausführt.

Weitere Aktionen definieren

Kindklassen von AbstractSyncKP haben die Möglichkeit weitere Aktionen zu definieren. Hierzu ist die Methode doAction zu überschreiben. Action selbst können vor dem Senden mit setAction gesetzt werden. AbstractSyncKP prüft zuerst seine eigenen Aktionen. Sollte die gesetzte Action keinem Pull oder Pullrequest entsprechen, so wird doAction. In der Standardimplementierung erzeugt dies einen Fehler, da die Klasse nicht weiß, was sie tun soll. Wenn die Methode überschrieben wurde, dann können hier weitere Aktionen geprüft werden. Es ist anzuraten immer super.doAction aufzurufen, wenn keine Zuordnung der Aktion möglich war.

4.2.2 Ein abstrakte Klasse als Basis

Um eine möglichst allgemeine Form zu wahren und das entwickelte Klassenkonstrukt für viele Fälle anwendbar zu machen wurde eine weitere abstrakte Klasse geschrieben, die von AbstractSyncKP erbt. Sie ist in der folgenden Abbildung 34 zu sehen.

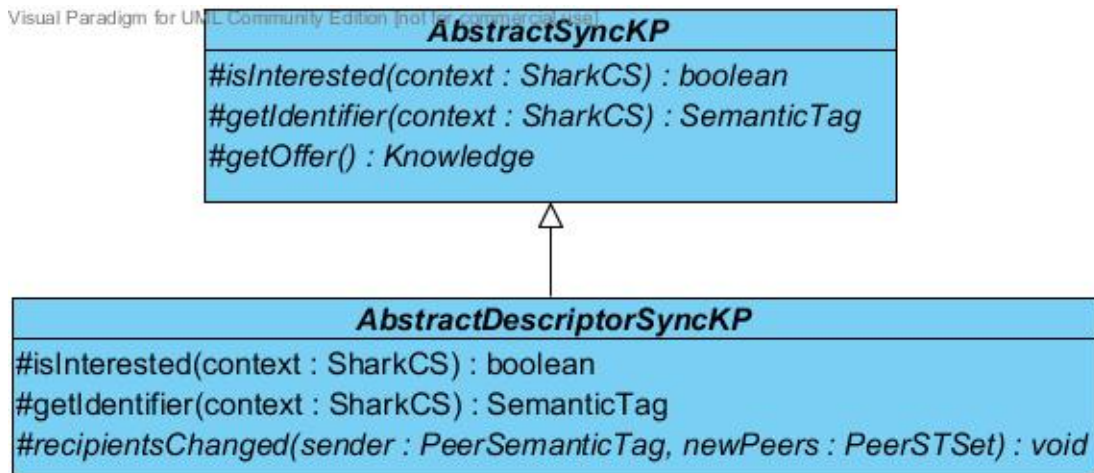


Abbildung 34: Klassendiagramm: AbstractDescriptorSyncKP

Die neue Klasse AbstractDescriptorSyncKP implementiert zwei der drei abstrakten Methoden von AbstractSyncKP. Dies bedeutet, dass Entwickler nur noch das extrahieren von Wissen übernehmen müssen. Die anderen Zwei funktionieren wie folgt:

- **Interesse bekunden:** Weitere Aktionen werden nur ausgeführt, wenn der gesendete Deskriptor gleich mit dem von der Klasse gehaltenen Deskriptor ist. Gleich bezieht sich hierbei auf die Gleichheit der Identifikatoren dieser.
- **Identifikatoren finden:** Der Identifikatoren des Knowledge Port ist ein künstliche Thema, das in der Topic Dimension des Interesse dieses angelegt wurde. Genauer ist in folgenden der Erklärung zum künstlichen Interesse beschrieben.

Nutzung eines künstlichen Interesses

Wie in Abschnitt 3.5.5 beschrieben wird ein künstlichen Interesse genutzt. Die Klasse AbstractDescriptorSyncKP baut daher besagtes Interesse anhand des übergebenen Deskriptor wie folgt auf.

- **Topics:** Die Topics Dimension besteht aus einem künstlich angelegten Thema. Dieses Thema dient zur dem Zweck Metadaten zu halten.
- **Peers:** Die Peers Dimension einhält den Eigentümer der Wissensbasis, auf welchem das Schema zum Deskriptor beruht.
- **Remote Peers:** Die Remote Peers Dimension wird entweder vom Konstruktor übernommen (siehe nachfolgende Erläuterung) oder, sollten keine Empfänger übergeben worden sein, entspricht der Remote Peers Dimension des Deskriptor, den diese Klasse hält.
- **Direction:** Die Direction Dimension entspricht initial immer INOUT. Eine Umschaltung zwischen INOUT und NOTHING ist zu jeder Zeit möglich.

Die anderen Dimensionen sind, entsprechend des Konzeptes, leer.

Konstrukturen von AbstractSyncKP und AbstractDescriptorSyncKP

In Abbildung 34 sind nicht die Konstrukturen der beiden Klassen dargestellt, um platz im Diagramm zu sparen. Dennoch sind diese von Bedeutung. Im folgenden die Parameter für die Konstrukturen der beiden Klassen.

AbstractSyncKP

- **SharkEngine sharkEngine:** Die SharkEngine, welche von dem Port benutzt wird.
- **SyncKB syncKB:** Die Wissensbasis, an der die Zeitstempel gespeichert sind.
- **SharkCS context:** Interesse dieses Ports. Die Remote Peers Dimension wird zu Erstellung der Zeitstempel genutzt.

AbstractDescriptorSyncKP

- **SharkEngine sharkEngine:** Die SharkEngine, welche von dem Port benutzt wird. Wird an AbstractSyncKP übergeben.
- **SyncDescriptorSchema schema:** Schema, in welchem sich der Deskriptor befindet.
- **ContextSpaceDescriptor descriptor:** Deskriptor, der den zu synchronisierenden Datenbereich beschreibt.
- **PeerSTSet recipients:** Personen, mit denen die Synchronisation stattfinden soll.

Zu erkennen ist, dass sich nur ein Parameter gleicht und direkt übergeben werden kann. Aus dem SyncDescriptorSchema kann die SyncKB gelesen und übergeben werden. Das künstliche Interesse wird anhand des Schemas, des Deskriptors und der Empfänger erstellt und dann ebenfalls dem Konstruktor von AbstractSyncKP übergeben. Damit einhält der dieser besagtes Interesse sowie die zum Schema gehörende Wissensbasis.

Besonderheiten von AbstractDescriptorSyncKP

Die Klasse AbstractDescriptorSyncKP weist einige Besonderheiten auf. Zum Einen definiert sie zusätzliche Aktionen, zum Anderen haben einige Methoden eine besondere Verhaltensweise.

- **Hinzufügen und entfernen von Adressaten:** Eine Änderung der Remote Peer Dimension am Interesse der Ports hat keine weiteren Aktionen zur folgen. Des müssen die Methoden addRecipient und removeRecipient genutzt werden, damit eine Notifikation der Änderung an die Kommunikationspartner erfolgt. Die Kindklasse von AbstractDescriptorSyncKP muss dabei implementieren, was in diesem Fall zu geschehen hat.
- **Änderungen Deskriptoren verteilen:** Es ist möglich Änderungen des aktuell beschriebenen Deskriptors zu verteilen. Dabei wird der gesamte Baum verteilt und gleiche Deskriptoren beim Ziel überschrieben.
- **Getter für den Deskriptor:** Der Getter liefert den Deskriptor direkt aus den Metadaten. Dies bedeutet, er wird aus dem XML erstellt. Somit schlagen sich Änderungen an diesem nicht sofort auf den Port aus. Um eine Änderung am Knowledge Port durchzuführen muss der Setter aufgerufen werden.
- **Subskription:** Die Methoden subscribe und unsubscribe setzen die Direction Dimension des Ports. Ein abonnierter Port wird immer Daten versenden und Empfangen, währenddessen ein nicht abonnierter Port keine Aktionen durchführt.

4.2.3 Eine Standardimplementierung

Mit StandardDescriptorSyncKP existiert eine Standardimplementierung, die den einfachsten Fall abdeckt. Die beiden noch zu implementierenden abstrakten Methoden führen folgende Aktionen aus.

- **Zusammenstellen des Angebotes:** Aus der Wissensbasis wird nur das Wissen zum Kontext des im Port gehaltenen Deskriptors extrahiert. Jede Art von Beziehungen findet keine Beachtung.
- **Änderung von Adressaten:** Die Remote Peer Dimension wird mit den gesendeten neuen Peers ausgetauscht. Dabei wird der Besitzer des Ports, genauer Besitzer des Schemas, welches am Port gespeichert ist, mit dem Sender ausgetauscht. Ist der Besitzer ein Teil der neuen Remote Peer Dimension, so bedeutet dies, dass der Sender der Nachricht mit diesem kommunizieren möchte. Ist der Besitzer nicht in der Liste der neuen Adressaten, so möchte der Sender nicht mehr mit diesem Port kommunizieren und wird daher aus den Adressaten entfernt. Abbildung 35 veranschaulicht den Ablauf noch einmal skizzenhaft in einem Aktivitätsdiagramm.

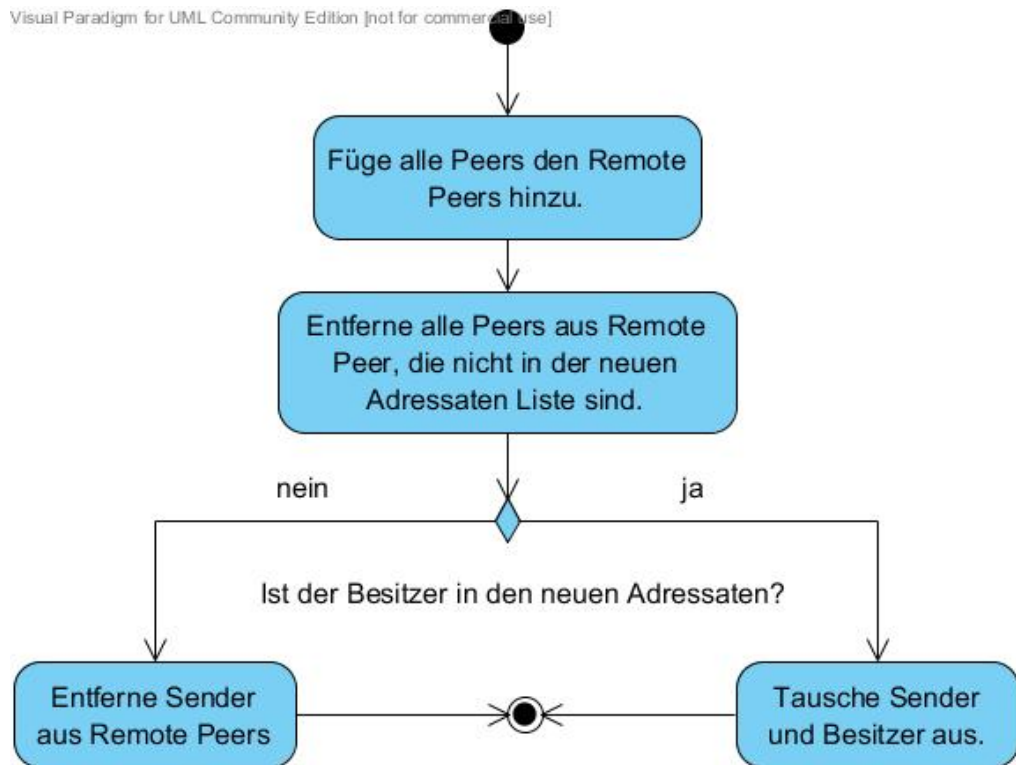


Abbildung 35: Skizze: Ablauf der Änderung von Adressaten

4.3 Ein Knowledge Port zum Lernen von Deskriptoren

Das Lernen und Lehren von neuen Deskriptoren übernimmt ein Knowledge Port, der völlig unabhängigen zu den bisher vorgestellten Ports operiert. Abbildung 36 skizziert die wichtigsten Aspekte der zugehörigen Klasse DescriptorAssimilationKP.

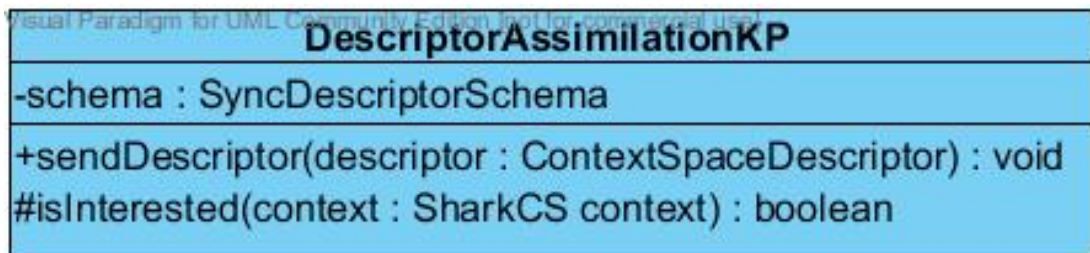


Abbildung 36: Klassendiagramm: DescriptorAssimilationKP

Die Klasse basiert auf einem Schema. Ein Port kann dabei beliebig viele Deskriptoren versenden. Voraussetzung aber ist, dass er sich so identisch im Schema befinden. Dabei wird nicht der Deskriptoren selbst gesendet sondern sein gesamter Baum. Dies ist notwendig, damit auch Beziehungen von Kommunikationspartnern erlernt werden. Die Adressaten fügen den Baum dann wie erhalten bei sich ein, wobei bereits existierende Deskriptoren überschrieben werden. Von Interesse ist auch die isInterested Methode. Im Standardfall liefert sie immer true zurück kann aber überschrieben werden um zu unterscheiden, wann ein Baum zu assimiliert ist und wann nicht.

5 Tests

Dieses Kapitel widmet sich den Test zur Qualitätssicherung. Es wird beschrieben, wie diese sichergestellt wurde.

5.1 Unit Test und Testabdeckung

Zur Sicherstellung der Funktionalität der Softwarekomponente wurden Unit Tests geschrieben. Da das Projekt ein Maven Projekt ist befinden diese sich, nach der Konvention, im Verzeichnis `src/test/java` (siehe Anhang A). Alle Test verlaufen ohne Fehler.

Des Weiteren wurde Cobertura genutzt um die Testabdeckung zu gewährleisten.

Package	# Classes	Line Coverage
All Packages	14	82% 566/683
net.sharkfw.descriptor.knowledgeBase	5	91% 214/235
net.sharkfw.descriptor.peer	4	79% 195/245
net.sharkfw.knowledgeBase.sync	2	73% 121/165
net.sharkfw.xml.jaxb	3	94% 36/38

Abbildung 37: Ausschnitt: Cobertura Report

Abbildung 37 zeigt einen Ausschnitt des Cobertura Reports. Zu sehen ist, dass die Line Coverage immer rüber 70% liegt und um Durchschnitt bei 83,8%. Dies wird als ausreichende Testabdeckung betrachtet.

5.2 Codequalität

Zur Sicherstellung der Qualität des Quellcodes wurde das Werkzeug PMD genutzt. Ein Report dazu kann auf der beiliegenden CD (siehe Anhang A) gefunden werden. Genutzt wurde die Standardkonfiguration von PMD. Das heißt die Regeln definiert in `basic.xml`, `unusedcode.xml` und `imports.xml` wurden überprüft. Die einzig auffallende Regelverletzung dabei ist *Useless parentheses*. Dabei handelt es sich im Code um die Zusammenstellung eine booleschen Ausdrucks. Die Klammern dienen der Lesbarkeit. Die *Avoid modifiers which are implied by the context* Warnung wird ebenfalls ignoriert, da sie für die Lesbarkeit als nicht relevant erachtet wird.

5.3 Test der Anforderungen

Im folgenden ist beschrieben, wie die Funktionale und nicht funktionale Anforderungen getestet wurden.

Funktionale Anforderungen

- **Beschreibbarkeit:** Die Beschreibbarkeit ist durch den Deskriptor umgesetzt worden. Es existieren Unit Test für das Schema von Beziehungen. Durch den fehlerfreien Ablauf dieser ist die Umsetzung dieser Anforderung gesichert.

- **Beziehungen:** Die Beziehungen wurden durch das Schema umgesetzt. Die hier existierenden Unit Tests sichern die Erfüllung dieser Anforderung.
- **Persistenz:** Über das Schema können Beschreibungen gespeichert und geladen werden. Unit Test beweisen, dass diese Anforderung umgesetzt wurde.
- **Synchronisation:** Es wurde ein Klassenkonstrukt geschaffen, welches eine Synchronisation auf der Grundlage der entwickelten Deskriptoren ermöglicht. Auch hier beweisen Unit Test deren Funktionalität.
- **Änderbarkeit:** Deskriptoren besitzen einen Identifikator über den sie Unabhängig von ihrem Kontext auffindbar sind. Somit kann eine Änderung ohne Verlust des Deskriptor durchgeführt werden. Beweis hierfür sind Unit Tests des Schemas.
- **Änderung kommunizieren:** Änderungen können über die Klassen zur Synchronisation kommuniziert werden. Diese wurden anhand von Modultest getestet und daher gilt die Anforderung als umgesetzt.

Nicht funktionale Anforderungen

- **Build-Management:** Maven wurde als Tool für das Build-Management gewählt. Das Projekt konnte in eine Netbeans IDE nach dem klonen aus einem Git Repository ohne Probleme importiert werden. Des Weiteren ermöglicht die erstellte Konfiguration das Erstellen von Cobertura und PMD Reports.
- **Testbarkeit:** Es existieren Unit Test, daher ist der Quellcode testbar.
- **Modultest:** Es wurden eine Reihe von Unit Test geschrieben und ein Cobertura Report erstellt.
- **Wartbarkeit:** Der PMD Report erzeugt nur unwesentliche Meldungen, die ignoriert werden können (siehe Erklärung in Abschnitt 5.2). Von daher kann davon ausgegangen werden, dass der Quellcode lesbar ist und somit gewartet werden kann. Für weitere Erklärungen kann die JavaDoc Dokumentation (siehe Anhang A) und diese Arbeit selbst herangezogen werden.

6 Fazit

In diesem Kapitel soll die Arbeit noch einmal in ihrer Gesamtheit betrachtet werden. Zuerst wird kritisch auf Mängel und Verbesserungsmöglichkeiten eingegangen. Anschließend folgt das Schlusswort, welches die Ergebnisse bewerte und für zukünftige Projekte einordnet.

6.1 Mängel und Verbesserungsmöglichkeiten

Folgend sind die Mängel und Verbesserungsmöglichkeiten der in dieser Arbeit entstandenen Softwarekomponente im Detail beschrieben.

- **Keine vollständige Synchronisation in einem Schritt:** Es ist nicht gelungen eine vollständige Synchronisation in einem einzigen Schritt zu implementieren. Das Ausführen der Aktionen Pull und Pull Request führt zwar zum gleichen Ergebnis, erfordert aber trotzdem, dass diese einzeln angestoßen werden. Leider ist im Shark Framework sowie im Algorithmus der SyncKB, auf dem aufgebaut wurde, nicht klar, wann ein Knowledge Port seine Arbeit beendet hat. Daher erfolgt eine vollständige Synchronisation in der entwickelten Softwarekomponente in zwei Schritten.
- **Kein Caching:** Die aktuelle Implementierung des Schemas ließt die Deskriptoren bei jedem zugriff aus ihrer persistenten Form. Dies können unter Umständen Dateizugriffe sein, was die Laufzeit der Aktionen erhöht. Auch die Umwandlung von XML in ein Java Objekt bei jedem Zugriff kostet Laufzeit. Ein Caching der Deskriptoren wäre anzuraten, sollte der Quellcode einem Refactoring unterzogen werden. In dieser Arbeit wurde in erster Linie auf Funktionalität Wert gelegt und das Laufzeitproblem deswegen vernachlässigt.
- **Keine Test von Multithreading:** Die vorhandenen Modultest beweisen die Funktionalität der Softwarekomponente nur im kleinen Rahmen. Test mit einer großen Anzahl an Personen, die sich synchronisieren wollen, wurden nicht durchgeführt. Das Verhalten in einer Situation dieser Art ist also unklar.
- **Komplexität:** Die Nutzung eines künstlichen Interesse ist komplex und entspricht nicht dem normalen Vorgehen im Shark Framework. Aufgrund der fehlenden Metadaten-Dimension war dies aber notwendig. Leider ist somit die interne Funktionsweise der Softwarekomponente für Außensehende schwer zu verstehen. Sollte je eine zusätzliche Dimension für Metadaten eingeführt werden, so ist ein Refactoring des künstlichen Interesses anzuraten.
- **Schema als Sammlung von Bäumen:** Das Schema ist aktuell eine Sammlung von Baumstrukturen. Grund hierfür ist, dass die meisten Social Media Formate entweder eine Liste von Einträgen sind oder selbst eine Baumstrukturen besitzen. In dieser Arbeit wurde nicht analysiert welche anderen Formate mit den Deskriptoren dargestellt werden können. Es ist möglich, dass in der Zukunft ein Netz oder eine Taxonomie besser geeignet ist um Beziehungen von Deskriptoren darzustellen.
- **Eliminierung des Piggyback Algorithmus:** Ob die Eliminierung des Piggyback Algorithmus gerechtfertigt war ist fraglich. Gerade bei dem Versand größerer Datenmengen, was hier nicht getestet wurde. Das Beheben des aufgetretenen Fehlers hatte Priorität. In zukünftigen Anpassungen sollte daher übergeleitet werden, ob der Piggyback Algorithmus wieder eingeführt wird.

- **Trennung von Assimilation und Deskriptoren-Synchronisation:** Die Klassen `AbstractDescriptorSyncKP` und `DescriptorAssimilationKP` können beide zum Austausch von Deskriptoren genutzt werden. Erstere behandelt nur Deskriptoren, die bereits bei allen Kommunikationspartnern existieren, währenddessen die Zweite das Erlernen dieser ermöglicht. Da sich Algorithmus und Aufbau ähneln ist zu überlegen, ob die Klassen vereint werden können.

6.2 Schlusswort

In dieser Arbeit ist es gelungen einen Algorithmus zu implementieren, der es ermöglicht definierte Bereiche einer Wissensbasis zu synchronisieren. Dabei ist eine Beschreibung eines Raumes von Daten die Grundlage. Diese ist nicht an die Synchronisation gebunden und kann auch für andere Anwendungsfälle genutzt werden. Es besteht die Möglichkeit Beziehungen zwischen den Räumen zu definieren. Besagte Beziehungen sind abstrakt und können je nach vorliegendem Fall einem anderen Zweck dienen. Durch Ableitung der Klassen zur Synchronisation ist es möglich diese anzupassen. Dadurch kann die vorliegende Softwarekomponente auch in Projekten mit anderen Anforderungen einzusetzen werden. Allerdings hat die Implementierung noch nicht ihre Tauglichkeit für größere Projekte bewiesen. Die durchgeführten Modultests sind klein bezogen auf die Anzahl an Personen und Daten. Es ist zu erwarten, dass in Zukunft Probleme auftauchen, die in dieser Arbeit nicht aufgegriffen wurden. Von daher stellt die Arbeit eher ein Proof of Concept als eine ausgereifte Implementierung da. Die Wahrscheinlichkeit, dass sich zukünftige Projekte mit den Verfeinerungen der hier beschriebenen Methode und Algorithmen befassen müssen, ist hoch.

7 Quellenverzeichnis

Artikel

- [1] Ulrich Clauß. „Europa sagt beim Datenschutz ”Yes, we can““. In: (6. Okt. 2015). URL: <http://www.welt.de/debatte/kommentare/article147296652/Europa-sagt-beim-Datenschutz-Yes-we-can.html> (besucht am 07.10.2015).
- [2] Hartmut Schlosser. „JAXenter - Java 9 ohne sun.misc.Unsafe: Ein Desaster?“ In: (17. Juli 2015). URL: <https://jaxenter.de/java-9-ohne-sun-misc-unsafe-ein-desaster-23130> (besucht am 03.10.2015).
- [3] Moritz Stückler. „t3n - Was ist eigentlich dieses GitHub?“ In: (15. Juni 2013). URL: <http://t3n.de/news/eigentlich-github-472886/> (besucht am 30.09.2015).

Bücher

- [4] Ellis Horowitz und Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1976. ISBN: 091489420X.

Handbücher

- [5] Prof. Dr. Thomas Schwotzer. *Building Semantic P2P Applications with Shark*. URL: http://www.sharksystem.net/sharkDeveloperGuide/Shark2.0_DevelopersGuide.pdf (besucht am 18.09.2015).

Publikationen

- [6] Scott A. Golder und Bernardo A. Huberman. *The Structure of Collaborative Tagging Systems*.
- [7] Peter Pin-Shan Chen. *The Entity-Relationship Model-Toward a Unified View of Data*. 1976.
- [8] Randall Reynolds Clemente Izurieta Nurzhan Nurseitov Michael Paulson. *Comparison of JSON and XML Data Interchange Formats: A Case Study*.
- [9] Mor Naaman und Chen Ye Oded Nov. *What Drives Content Tagging: The Case of Photos on Flickr*. 2008.

Spezifikationen

- [10] Joe Fialli Kohsuke Kawaguchi Sekhar Vajjhala. *The Java Architecture for XML Binding (JAXB) 2.2*. JSR-222. 4150 Network Circle Santa Clara, CA 95054 USA: Sun Microsystems, Inc., 10. Dez. 2009.

Webseiten

- [11] EJP. *Antwort auf Frage: Official reasons for Software caused connection abort: socket write error*. URL: <http://stackoverflow.com/questions/2126607/official-reasons-for-software-caused-connection-abort-socket-write-error/7747693#7747693> (besucht am 06.10.2015).

- [12] *GitHub*. URL: <https://de.wikipedia.org/wiki/Dateisystem> (besucht am 30.09.2015).
- [13] *Github SharkFW - SyncKB*. URL: <https://github.com/SharedKnowledge/SharkFW/tree/master/src/java/core/net/sharkfw/knowledgeBase/sync> (besucht am 21.09.2015).
- [14] Stefan Hagen. *Dateisystem - Wikipedia*. URL: <https://de.wikipedia.org/wiki/Dateisystem> (besucht am 30.09.2015).
- [15] *JavaDoc - XmlAdapter*. URL: <https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html> (besucht am 04.10.2015).
- [16] *Jenkins*. URL: <http://jenkins-ci.org/> (besucht am 15.09.2015).
- [17] *opendatastructures.org - 12. Graphs*. URL: http://opendatastructures.org/versions/edition-0.1g/ods-python/12_Graphs.html (besucht am 28.09.2015).
- [18] *opendatastructures.org - 3.1 SLList: A Singly-Linked List*. URL: http://opendatastructures.org/versions/edition-0.1g/ods-python/3_1_SLList_Singly_Linked_Li.html (besucht am 26.09.2015).
- [19] *opendatastructures.org - 3.2 DLList: A Doubly-Linked List*. URL: http://opendatastructures.org/versions/edition-0.1g/ods-python/3_2_DLList_Doubly_Linked_Li.html (besucht am 26.09.2015).
- [20] Prof. Dr. Thomas Schwotzer. *Shark framework*. URL: <http://www.sharksystem.net/> (besucht am 31.08.2015).
- [21] *Skype*. URL: <http://www.skype.com/de/home/> (besucht am 30.09.2015).
- [22] *stackoverflow*. URL: <http://stackoverflow.com/questions/32942615/how-to-convert-dot-separated-string-into-array-list> (besucht am 05.10.2015).
- [23] Paul E. Black und. *Dictionary of Algorithms and Data Structures*. URL: <http://xlinux.nist.gov/dads/HTML/tree.html> (besucht am 26.09.2015).
- [24] *WoltLab GmbH - Burning Board Community Forum*. URL: <https://community.woltlab.com/> (besucht am 30.09.2015).

8 Abbildungsverzeichnis

1	Shark Context Space Modell	5
2	Kommunikation der SyncKB	8
3	Algorithmus zum Extrahieren und Einfügen von Wissen der SyncKB	9
4	Aufbau und Operation von Einfach-Verkettete-Listen. Quelle: [18]	10
5	Aufbau von Zweifach-Verkettete-Listen. Quelle: [19]	10
6	Aufbau eines Baums. Quelle: [23]	11
7	Aufbau eines Graphen. Quelle: [17]	11
8	Vereinfachte Skizze des Entity Relationship Modell	12
9	Aufbau Chat in Skype. [21]	12
10	Forumstruktur: Oberste Ebene. Quelle: [24]	13
11	Forumstruktur: Thread-Sammlung Ebene. Quelle: [24]	13
12	Forumstruktur: Thread Ebene. Quelle: [24]	14
13	Illustration über den Vergleich von Dateisystem-Bäumen. Quelle: [14]	15
14	Illustration von Tags an einer Frage in stackoverflow. Quelle: [22]	15
15	Scenario 1 JSON vs. XML Timing. Quelle: [8]	17
16	Scenario 1 JSON vs. XML CPU/Mem. Quelle: [8]	17
17	Scenario 2 JSON Vs XML Timing. Quelle: [8]	18
18	Scenario 2 JSON CPU/Mem. Quelle: [8]	18
19	Scenario 2 XML CPU/Mem. Quelle: [8]	19
20	Konzeption eines Deskriptor	22
21	Skizze: Selbstreferenz	24
22	Beziehungen von Deskriptoren für Chats	24
23	Beziehungen von Deskriptoren eines Forum	25
24	Beziehungen von Deskriptoren in einem Dateisystem	26
25	Skizze: Synchronisation Algorithmus der aktuellen SyncKB	28
26	Vereinfachter Algorithmus zur Synchronisation	29
27	Skizze: Künstliches Interesse für partielle Synchronisation	31
28	Skizze: Abstraktionsebenen	32
29	Klassendiagramm: Deskriptor und Schema	34
30	Skizze: Factory Pattern für JAXB Serialisierung	37
31	Skizze: Algorithmus Extraktion	38
32	Klassendiagramm: AbstractSyncKP	39
33	Ablauf einer Synchronisation	39
34	Klassendiagramm: AbstractDescriptorSyncKP	41
35	Skizze: Ablauf der Änderung von Adressaten	44
36	Klassendiagramm: DescriptorAssimilationKP	44
37	Ausschnitt: Cobertura Report	45

A CD-ROM zur Arbeit

Die beiliegende CD-ROM zur Arbeit enthält die folgenden Inhalte.

- **Masterarbeit.pdf:** Diese Arbeit im PDF Format.
- **Verzeichnis Quellcode:** Der Quellcode zu dieser Arbeit als ein Maven-Projekt. Das Projekt bringt alle Abhängigkeiten mit und kann über den *install* Befehl von Maven gebaut werden.
Weitere Informationen zu maven sind unter <https://maven.apache.org/> zu finden.
- **Dokumentation:** JavaDoc Dokumentation des Quellcodes, sowie ein Cobertura und PMD Report zur Qualitätssicherung.

Die Quellen existieren zusätzlich in GitHub unter <https://github.com/SharedKnowledge/Incubator/tree/master/Descriptor>.

Das Maven Projekt ist konfiguriert eine Cobertura Report über den Maven Befehl *cobertura:cobertura* zu erstellen. Ein PMD Report kann über den Maven Befehl *pmd:pmd* erstellt werden.

B Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum: Berlin, 12. Oktober 2015

Unterschrift: