Masterarbeit

Ein System zur partiellen Synchronisation von Wissensbasen für dezentrale soziale Netzwerke

von Jens Grundmann
5. 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

Soziale Netzwerke erfreuen sich immer mehr Beliebtheit in den letzten Jahren. Sei es Facebook oder schlicht das Forum zum Online Game, dass 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 bedeutetet, dass alle Daten auf einem entfernten Server gespeichert sind. Der Nutzer hat somit keine Kontrolle bzw. nur die beschränkte Kontrolle, welche der Betreiber des sozialen Netzwerkes anbietet, wie mit seinen Daten umgegangen wird.

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 benötige, welche die Daten der lokalen Systeme mit einander synchronisiert.

Ziel dieser Arbeit ist es eine Softwarekomponente zu entwickeln, die eine partiellen Synchronisation von Wissensbasen ermöglicht. Eine Wissensbasis ist dabei nichts anderes als eine Menge an Daten, die in einer bestimmten Stuckatur 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 [18] von Prof. Dr. Thomas Schwotzer und die darin enthalte SyncKB Klassensammlung. Sie ermöglicht bereits eine Synchronisation aller Daten in einer Wissensbasis. Diese soll nun dahingehen ausgebaut werden, dass Teile der Wissensbasis beschrieben werden können und nur diese beschrieben Teile synchronisiert werden.

Zuerst muss mit 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 regieren 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 der Funktionalität der Softwarekomponente wird schließlich eine Chat mit grafisch Oberfläche geschrieben, in dem einzelne Peers sich miteinander unterhalten können.

Die Arbeit wird zuerst die Grundlagen, wie das Shark Framework [18], auf denen die Implementierung beruht, erläutern. Danach wird das Konzept der Softwarekomponente erstellt. Es werden verschieden Möglichkeit diskutiert, um die im letzten Absatz beschriebenen Anforderungen umzusetzen. Im Anschluss wird die eigentliche Implementierung vorgestellt und auf diese eingegangen. Nachfolgend werden die Tests und Methoden nur Qualitätssicherung gezeigt und erklärt. Zuletzt wird ein Fazit gezogen sowie ein Ausblick auf mögliche Verbesserungen oder Erweiterungen.

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 Framesworks und Modelle besprochen, die zur Entwicklung der Komponente und Umsetzung der Anfordern genutzt werden können.

2.1 Funktionale und nicht funktionale Anforderungen

Im Folgendem werden die funktionalen und nicht funktionale 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:** Es ist möglich Beziehungen zwischen Räumen zu definieren. So soll beispielsweise der Raum Java-Chat ein Kind des Programmiersprachen-Chat Raumes sein können.
- **Persistenz:** Es soll möglich sein die Beschreibung der Räume von Daten persistent zu speichern. Die gespeicherten Räume bleiben somit erhalten und können zu späterem Zeitpunkt neu geladen werden.
- **Synchronisation:** Es ist möglich die Räume von Daten und ihre Abhängigkeiten mit Peers in einem Peer to Peer Netzwerk zu synchronisieren. Ziel ist es, dass die Räume nach der Synchronisation identisch von Aufbau und Inhalt sind.
- **Änderbarkeit:** Es muss möglich sein eine Beschreibung eines Raumes zu ändern inklusive ihrer Abhängigkeiten. Der Raum selbst soll dabei bestehen bleiben und ohne weiteren Aufwand auffindbar wie zuvor. Das heißt, er muss genauso aus der seiner persistenten Form geladen werde können wie zuvor 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 [14], sind wünschenswert.
- **Testbarkeit:** Die zu Softwarekomponente ist modular so aufgebaut, dass sie durch Modultest testbar ist.
- **Modultest:** Es existieren bereit eine Reihe von Modultest, 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 [18] eingegangen, die für die Arbeit benötigt werden. Das gesamte Framework wird nicht erklärt. Weiterführende Informationen sind im Developer Guide [4] 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 ist eine Beschreibung was der Kontext bedeutet.
- **Orginator:** Der Autor des beschriebenen Kontextes. Dies kann entweder der Ersteller sein oder der Autor der beiliegen Informationen. Bei einem wissenschaftlichen Artikel wäre dies der Autor des besagten Artikels.
- **Peers:** Die Peers Dimension kann unterschiedlich interpretiert werden. Zum einen kann es der Ersteller des Kontextes sein, der sich vom Autor der beiliegenden Informationen unterschieden kann. Zum anderen kann es der Besitzer des Kontextes sein, wobei Besitzer hier je nach beiliegendem Fall anders interpretiert werden kann. 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 biete die Möglichkeit ein Zeitintervall anzugeben.
- **Locations:** Gibt einen Ort an. Diese Information kann je nach vorliegendem Fall anders interpretiert werden.
- **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 beachte ist, dass wenn von einem Kontext gesprochen wird, dann 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. Wehrendessen besitzt der Kontextpunkt nur je genau einen Inhalt. Beispielhaft beschriebt der Kontextpunkt nur das Thema Java, während der Kontext Java und C beinhaltet. Die Dimensionen selbst werde 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 bildeten 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 dieser, 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 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. Wenn ein Interesse versandt wurde, so wird sie als erstes aufgerufen. Ihre generelle Aufgabe ist 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 KEPConnection Objekt mit dem sie Informationen über den Sender erhalten können. Des Weiteren können mithilfe dieses Objektes Daten an die doExpose und doInsert-Methode anderer Peer gesendet werden. Dies erlaubt einen mehrfachen Datentausch. Ein Beispiel hierzu ist der Synchronisationsprozess der SychKB, der im Nächten Abschnitt beschrieben wird und in Abbildung 2 skizziert ist.

Zu beachten ist, dass die Aussagen hier dem Standardfall entsprechen, 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 generelle Funktionsweise der SyncKB. Daher soll hier auf diese dieser eingegangen werden. Weitere Informationen können dem Quellcode selbst entnommen werden, der im Github vom SharkFW zu finden ist. [11]

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 basiert hauptsächlich auf dem Entwurfsmuster Wrapper. So wurden den einzelnen Element zwei zusätzliche Informationen mitgeben:

- **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.

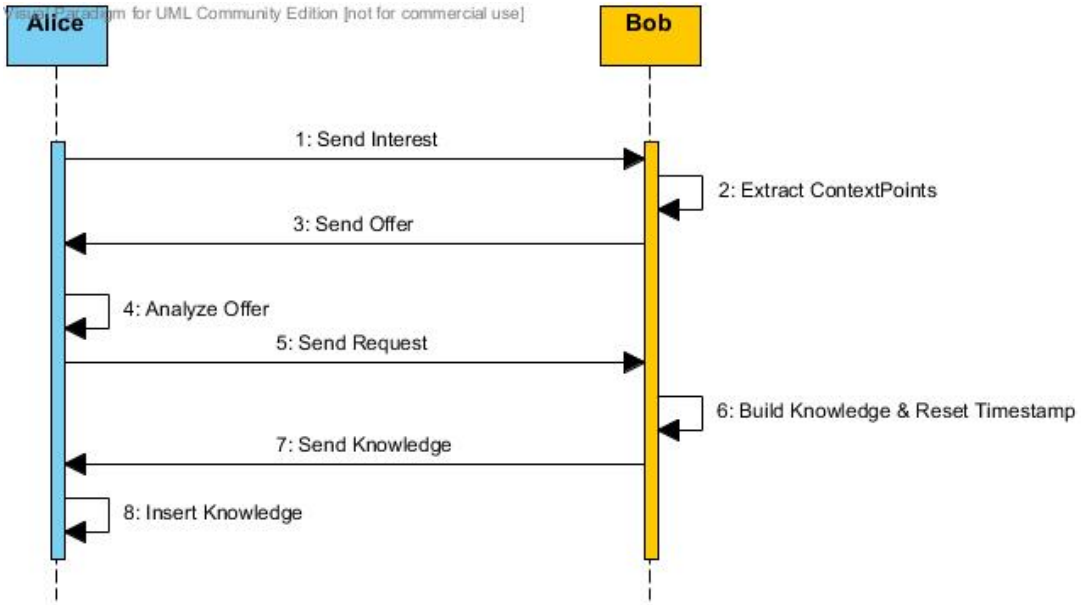

Abbildung 2: Kommunikation der SyncKB

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 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:** Die anzufragenden Kontextpunkte werden abermals am künstlichen Interesse per Property gespeichert und Bob zugesandt.

6. **Wissen bauen und Zeitstempel setzen:** Die angefragten Kontextpunkte werden nun aus Bobs Wissensbasis extrahiert und in einem Knowledge Objekt gespeichert. 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 das im rechten Teil von Abbildung 3 skizzierten Algorithmus und fügt diese dann in ihre Wissensbasis ein.

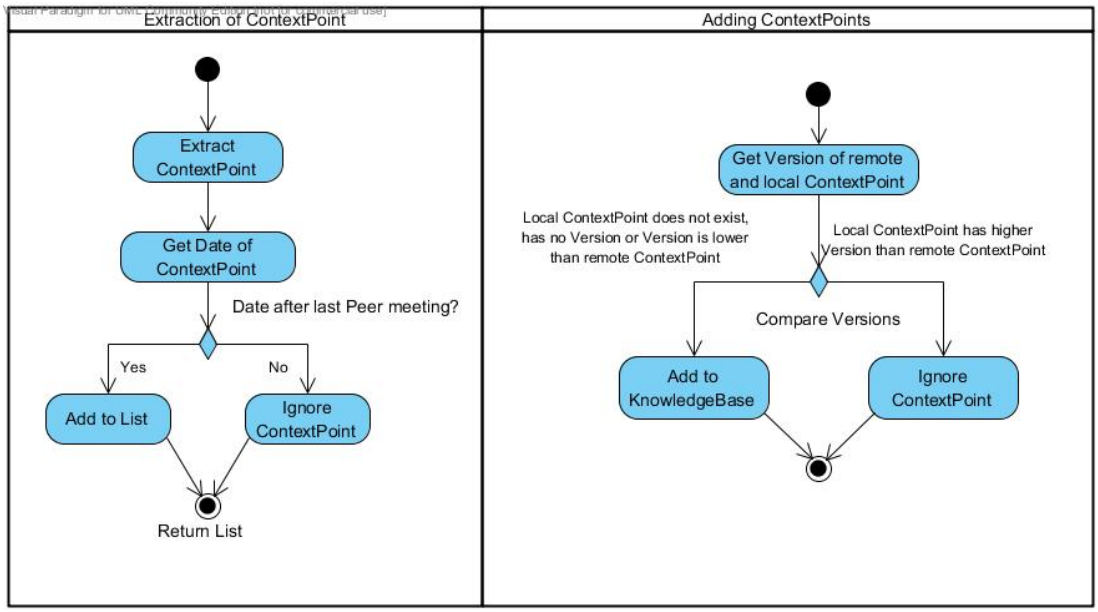

Abbildung 3: Algorithmus zum Extrahieren und Einfügen von Wissen der SyncKB

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 vollständiges 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 darstellen können. Dabei wird darauf eingegangen, wie sie die Daten und Beziehungen untereinander dargestellt sind. Weitergehende Erklärungen, wie mögliche Operationen, werden nur durch Links zu entsprechender Literatur gegeben.

2.3.1 Verkette Listen

Verkette Listen (beschrieben in [3], Kapitel 4) sind Listen, wo jedes Element Referenzen auf weitere Mitglieder der Liste einhält. Hier sollen die Einfach-Verketteten-Listen und die Zweifach-Verketten-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. Weitere Informationen können der Erklärungen unter [16] entnommen werden.

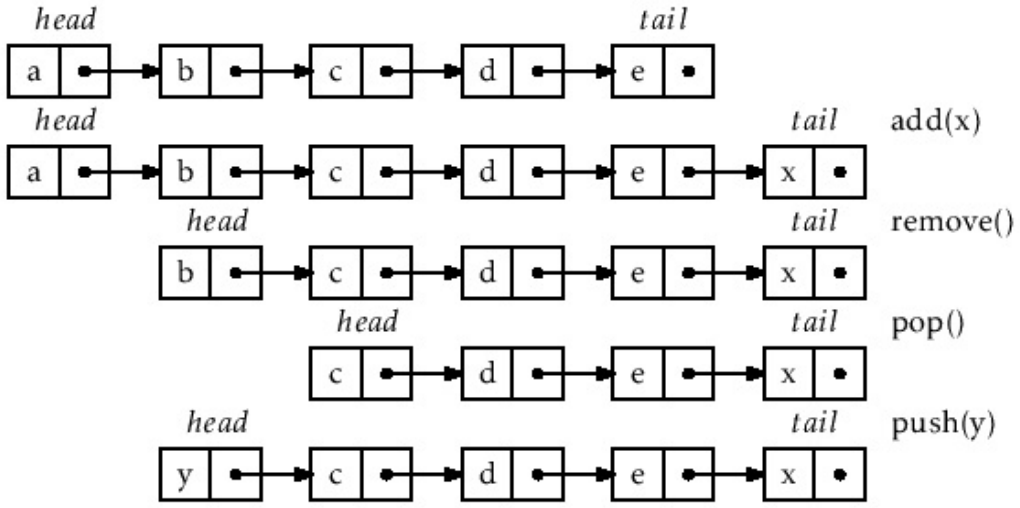

Abbildung 4: Aufbau und Operation von Einfach-Verkettete-Listen. Quelle: [16]

Zweifach-Verkette-Listen

Zweifach-Verkette-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 [17] entnommen werden.

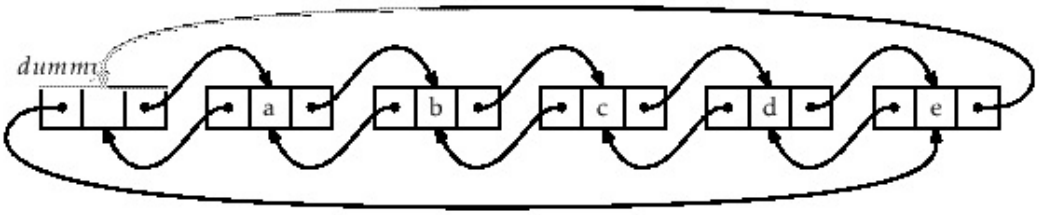

Abbildung 5: Aufbau von Zweifach-Verkette-Listen. Quelle: [17]

2.3.2 Bäume

Bäume ([3], 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.

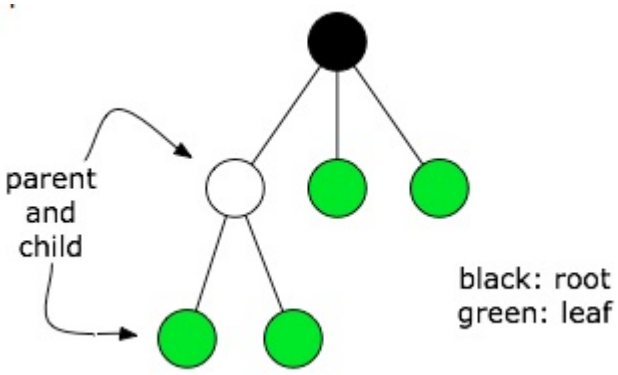

Abbildung 6: Aufbau eines Baums. Quelle: [21]

2.3.3 Graphen

Graphen (siehe [3], 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. Gerichtet Graphen geben dieser Beziehung eine Richtung. Somit kann allerdings nur angeben 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 in gewichteten Graphen Metadaten, wie die Kosten für einen Übergang von Element A zu Element B, den Richtungen mitzugeben. Dennoch bleibt die Art wie zwei Elemente im Detail zueinander in Beziehung stehen umbeschrieben. Abbildung 7 skizziert das Modell eines gerichteten Graphen. Die Peile stellen dabei die Übergänge zwischen den Elementen da. Weitere Erklärungen könne unter [15] gefunden werden.

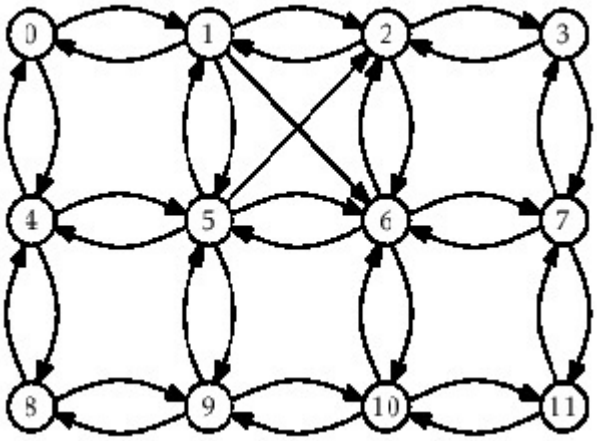

Abbildung 7: Aufbau eines Graphen. Quelle: [15]

2.3.4 Entity Relationship Modell

Das Entity Relationship Modell, beschrieben in [6], ist weniger eine Datenstruktur, als ein Modell zur Beschreibung von Zusammenhängen zwischen Daten bekannt aus relationalen Datenbanksysteme. Damit beschriebt es aber auch eine Beziehung und kann somit als Grundlage herangezogen werden. Die Beziehungen basieren darauf, dass jedes Element einen eindeutige 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 mehre 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.

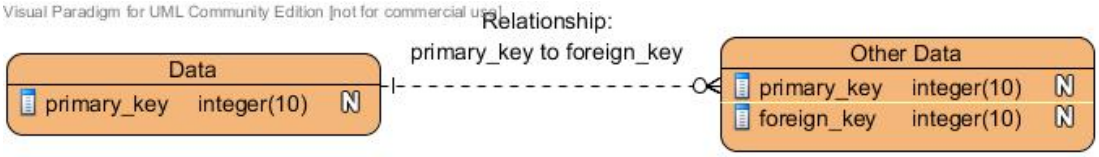

Abbildung 8: Vereinfachte Skizze des Entity Relationship Modell

2.4 Aufbau von Social Media Formaten

In diesem Abschnitt soll der Aufbaue eine Reihe von Social Media Formaten beschrieben werden. Social Media Formaten 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.

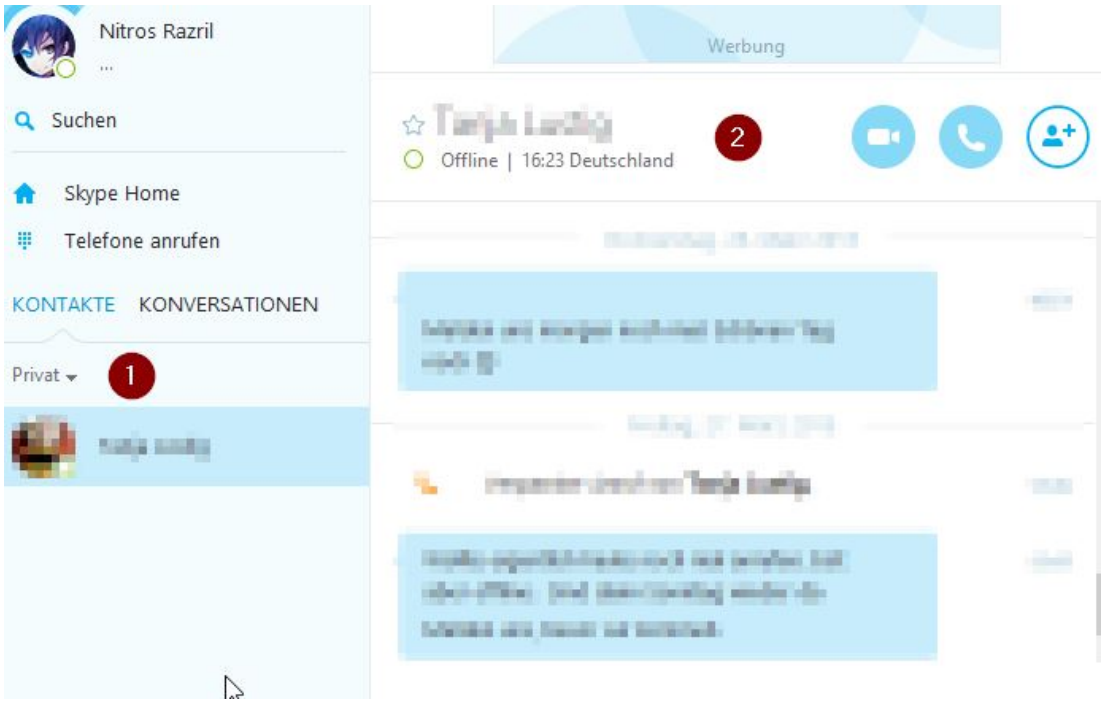

Abbildung 9: Aufbau Chat in Skype. [19]

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 einhält 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 eine einzelner Raum von Informationen ist, der auf eine bestimmte Art durch einen oder mehre Identifikatoren dargestellt wurde.

2.4.2 Forum

Ein Forum ist gegenüber einem Chat meist komplexer und lässt sich in mehr Ebenen einteilen. Als Beispiel soll das Burning Board der WoltLab GmbH herangezogen werden [22]. In den folgende Abbildungen 10, 11 und 12 ist der strukturelle Aufbau eines Forums zu sehen.

Abbildung 10: Forumstruktur: Oberste Ebene. Quelle: [22]

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) ausgehen können.

Abbildung 11: Forumstruktur: Thread-Sammlung Ebene. Quelle: [22]

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

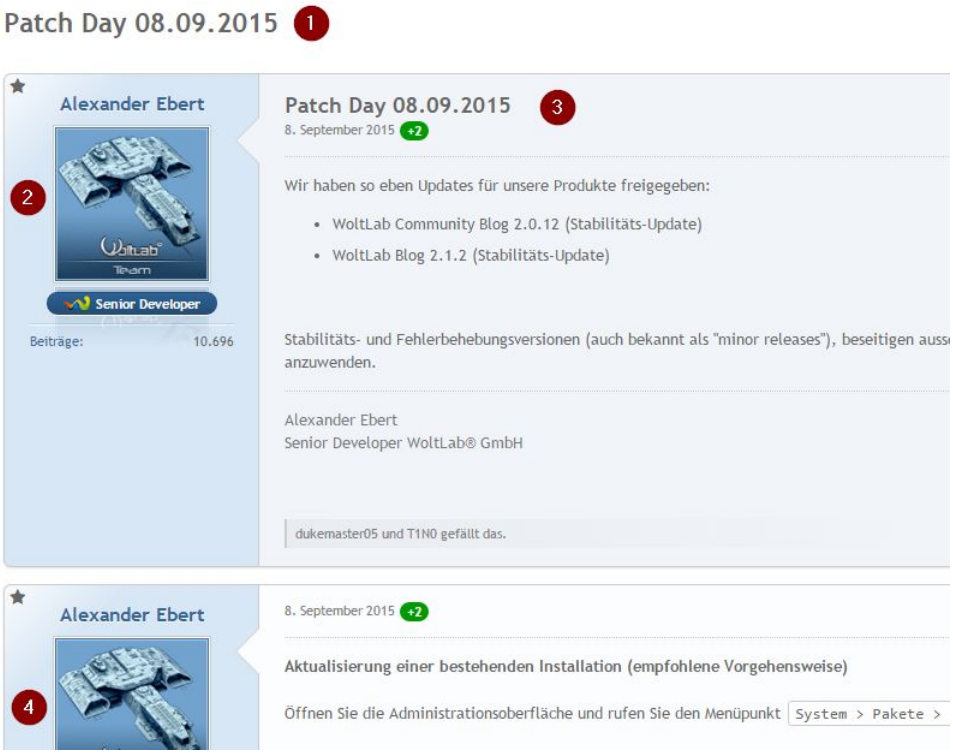

Abbildung 12: Forumstruktur: Thread Ebene. Quelle: [22]

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 die 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 [2] wird beschrieben, dass man immer mehr auf den webbasierter Filehosting-Dienst für Software-Entwicklungsprojekte GitHub [10] 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 Dateisystem für unterschiedliche Betriebssysteme.

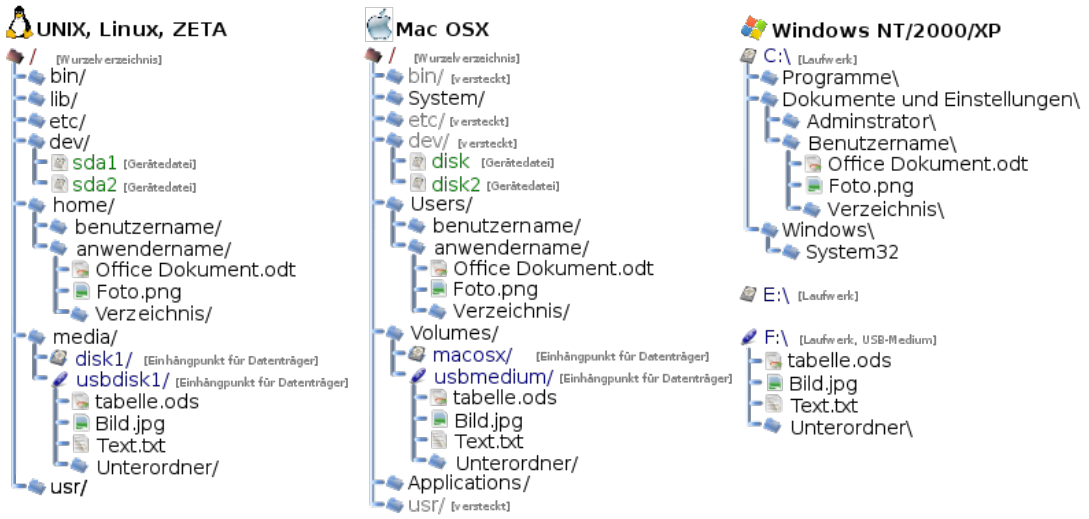

Abbildung 13: Illustration über den Vergleich von Dateisystem-Bäumen. Quelle: [12]

Zu sehen ist, das identisch zum Forum eine Baumstruktur die Grundlage von Dateisystemen ist. Hier enthalten nur die Blätter die eigentlichen Daten und die anderen Knoten diene lediglich der Eingruppierung von Daten. Die Blätter selbst können als ein einzelnes Element bestehend aus Bytes angereichert mit Metadaten gesehen werden. Dies unterscheidet sich zum Chat oder Forum, wo ein Blatt einem 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: [20]

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 er Grund zur Nutzung von Tags? Oded Nov, Mor Naaman und Chen Ye beschrieben es in ihrer Publikation What Drives Content Tagging: The Case of Photos on Flickr [8] als eine Möglichkeit Inhalten mit Metadaten, in der Form von Schlüsselwörtern, anzureichern. Der Nutzer nutzt Tags, damit seine Inhalte besser von anderen gefunden werde. 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 [5] 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? Zum Beispiel ein Buch oder ein Artikel 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 allgemeinem 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 Zugrodung 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 Anfordern. Zum Vergleich sollen die Ergebnisse aus der Fallstudie [7] zu rate gezogen. Diese sind 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: [7]

	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: [7]

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: [7]

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: [7]

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: [7]

Die Abbildungen 17, 18 und 19 zeigen den zweiten Fall der Fallstudie. Hier wurden mehrfach hintereinander kleine Datensatze überragen 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, was eine gesonderte Sichtweise erfordert.

- **Abhängigkeiten:** Umgangssprachlich gibt es den Begriff der "Jar-Hölle". Dieser beschriebt, 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 dieses durch das Nutzen eines Tools wie Maven verhindert werden, allerdings nutzt das Shark Framework dieses 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 dieses 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-Shema 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 gewinne 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 [1] 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 in 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. Wehrendessen einhält die Synchronisation die eigentliche Logik zum Ausführen der partiellen Synchronisation.

3.3 Ein Nachschlagewerk für Datenbereiche

Wie in Abschnitt 2.5 beschrieben eigenen sich Tags zum organisieren, filtern und suchen von Inhalten. Die Datenbereiche, die beschrieben werde 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 Abhängigkeit auffindbar und Datenbereiche könnten entsprechend erkannt werden. Die Idee ist hierbei ähnlich der Themen und Peer Taxonomie, welche bereist 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 Datenbereiches dargestellt werde 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, sodass eine Struktur ähnlich der Social Media Formate erreicht werde 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 die Analyse der in Abschnitt 2.3 beschrieben Datenstrukturen im Bezug auf die in Abschnitt 2.4 genannten Social Media Formate.

Verkettete-Listen

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

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

Bäume

Die meisten der besprochenen Social Media Formaten 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 hingegn ist die Tatsache, dass in der Programmiersprache Java, in welcher die Softwarekomponente geschrieben wird, keine vorgefertigte Datenstuck hierzu enthalten ist.

Graphen

Bäume sind prinzipiell eine besondere Art von Graphen. Daher lassen sich die Social Media Formaten 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 seinem Kindern kommt und das ein Kind immer seinen Vater kennt. Bäume sind für diesem Aspekt ausreicht, besonders da sie einen Startpunkt, die Wurzel, besitzen.

Entity Relationship Modell

Auch wenn eher ein Modell als eine Datenstruktur, so ist ein Element im Entity Relationship Modell eindeutig durch seinen Primärschlüssel beschrieben. Eine Beziehung zu einem Anderen lässt sich hier einfach durch einen Fremdschlüssel erreichen. Dies ermöglicht die Flexibilität, dass ein Element sehr leicht mehre Kinder und Väter haben. Auch als 1:1, 1:N. N:M Beziehung aus Datenbanksystemen bekannt. Viele Elemente der Social Media Formaten sind durch einen Identifikator identifizierbar. Beipeilweise 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. Wir haben im Rahmen dieser Arbeit keine relationale Datenbank mit einem Datenbanksystem, welche dies für uns übernehmen könne.

Auswertung

Da die beschriebenen Social Media Formate bereits eine 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. Serialisiert kann das Ganze als eine Liste 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 Datenbereiches beschrieben wird. Abbildung 20 zeigt den Aufbau unseres beschreibenden Elements, dem Deskriptor.

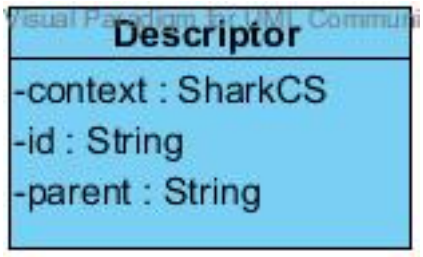

Abbildung 20: Konzeption eines Deskriptor

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 zuordnen. 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 wieder auffinden 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 Aufwinden 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 eine technische Identifikator, der von Maschinen und nicht Menschen verwendet wird, handelt. Die Darstellung als Zeichenkette dient jegliche 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 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 mehre 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:** Deskriptor sind gleich, wenn ihre Identifikatoren gleich sind. Das heißt, die Zeichenketten müssen übereinstimmten.
- **Ein Deskriptor ist *identisch* zu einem Anderen:** Deskriptor sind identisch zueinander, wenn alle ihre Elemente gleich sind. Das heißt, die Zeichenketten des Identifikator und Vater müssen mit denen, eines anderen Deskriptor übereinstimmten. Des Weiteren muss der Kontext, der beiden Deskriptoren, nach Regeln des Shark Frameworks, identisch sein.

Diese Unterscheidung ist notwendig, um das wiederzufinden eines Deskriptor sicher zu stellen und doppelte Datenhalten zu verhindern. Anhand des Identifikators kann nach einem bestimmten Deskriptor gesucht werde. Dennoch muss es möglich sein Unterschiede in den Parametern Vater und Kontext zu erkennen für mögliche Anpassungen. 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 nötigen, 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 einhält. 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 Eigenschaften, die einem Deskriptor ermöglichen eine Beziehung darzustellen, wurde bereits im vorhergehenden Abschnitt erklärt. Das Grundprinzip ist eine Selbstreferenz, die in Abbildung 21 dargestellt ist.

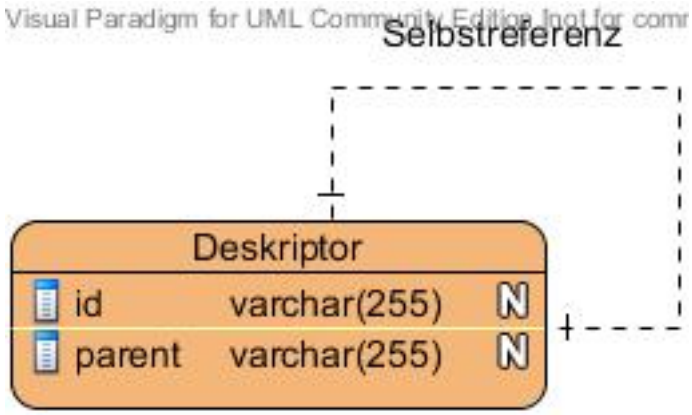

Abbildung 21: Skizze: Selbstreferenz

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. Dies erlaubt uns einen Baum von beliebiger Tiefe zu erstellen. Ein Vater kann auch auch beliebig viele Kinder haben, währendes ein Kind nur einen Vater besitzen kann.

Folgenden soll die Anwendung dieses Modells auf die in Abschnitt 2.4 besprochenen Social Media Formaten 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.

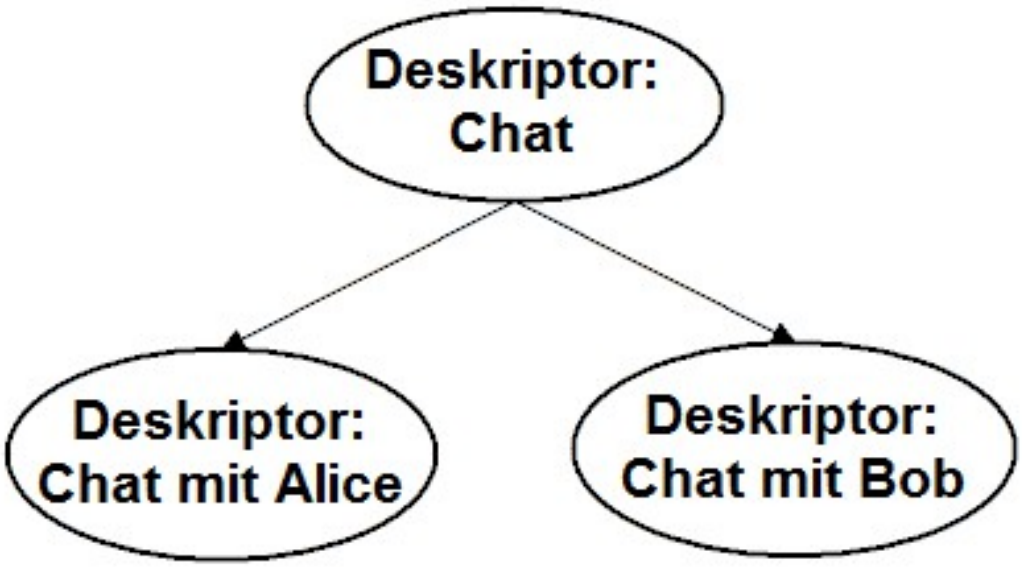

Abbildung 22: Beziehungen von Deskriptoren für Chats

Hier existiert genau eine Vater-Kind Beziehung. Dabei ist die Wurzeln 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 Chat findet. Das Vaterelement soll hierbei keine Informationen oder Kontextpunkte beschrieben. Es dient lediglich der Zuordnung. Der Kontext kann somit als leer angesehen werden. Alle Kindelemente des Deskriptor Chat hingeben 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.

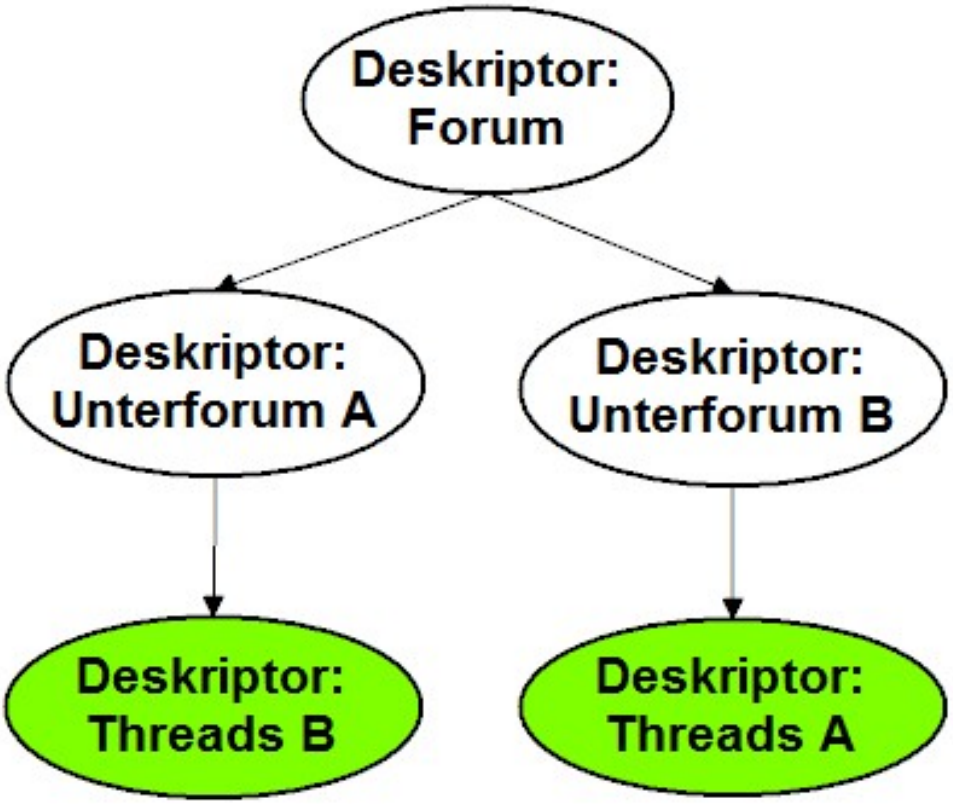

Abbildung 23: Beziehungen von Deskriptoren eines Forum

Abgebildet wird 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 schrieben immer Threads. Ein nicht leerer Deskriptor mit einer Beziehung beschreibt Threads und Unterforen.

Die Struktur von Dateisystem ist gleich der eines Forum. Abbildung 24 zeigt die Beziehungen für Verzeichnisse und Dateien.

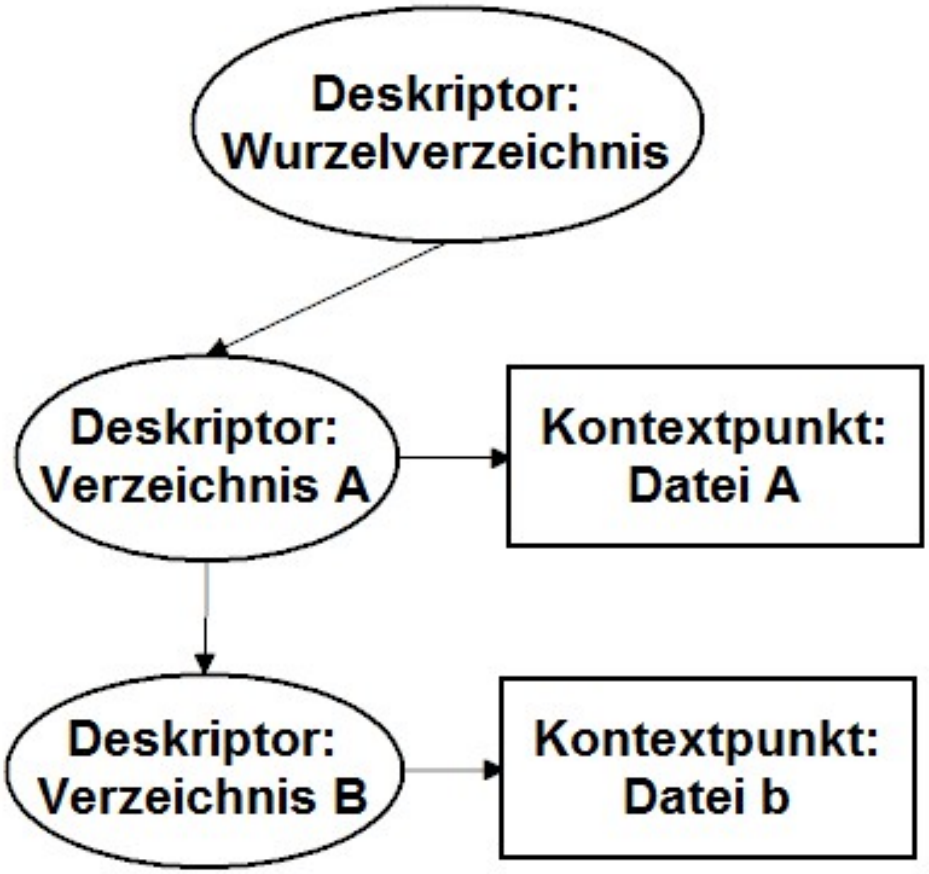

Abbildung 24: Beziehungen von Deskriptoren in einem Dateisystem

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 mehre 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 bereist 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 beschrieben 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.

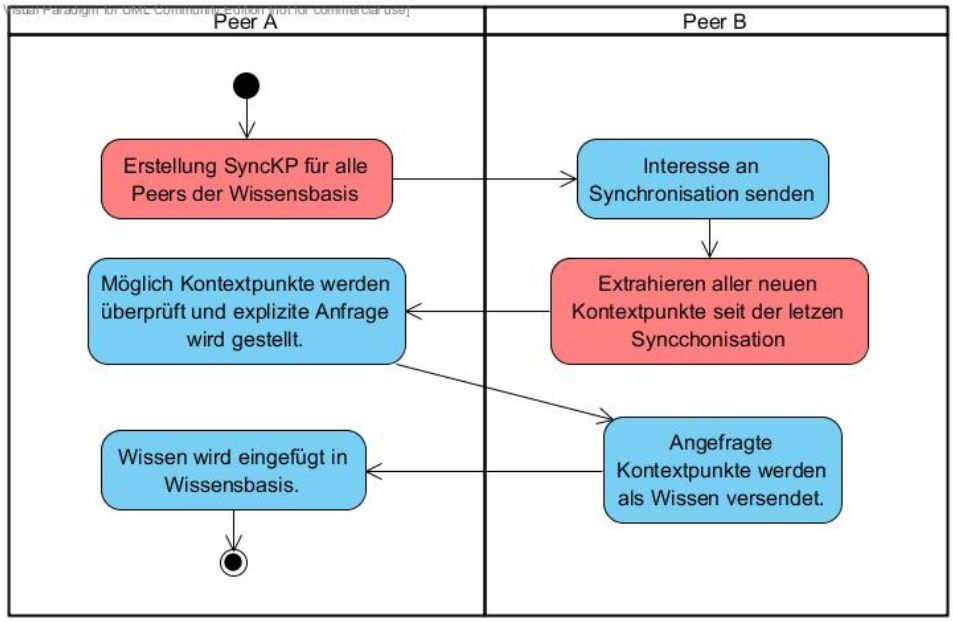

Abbildung 25: Skizze: Synchronisation Algorithmus der aktuellen SyncKB

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.

3.5.2 Abstraktion von der SyncKB

Aufgrund der im letzten Abschnitt beschrieben 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 mehre Möglichkeiten Wissen anhand eines Deskriptor zu extrahieren. Daher ist es sinnvoll dem Entwickler zu überlassen wie genau das Wissen aus der Wissensbasis extrahiert wird.

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 26 zeigt das Klassendiagramm beider.

Abbildung 26: 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 überschieben equals Methode pürft 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 Framesworks 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 hashSet 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 Gleicheit 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 Überschrieben 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 neue 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 Baumstatur 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 dargestellt 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 das im Java Development Kit enthalten JAXB zum Serialisieren des Schemas verwendet werden. Hierzu sind die Klasse ContextSpaceDescriptor mit entspreche Annotations ausgestattet. Für den Parameter context wurde eine entsprechender XmlAdapter (siehe [13]) geschrieben, der die Serialisierung einer SharkCS Klasse unter Zuhilfenahme der im Shark Framework existierenden Klasse XMLSerializer für JAXB ermöglicht.

Aufgrund der Annotations wurde auch drauf 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 [9] 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 27 skizziert dieses.

Abbildung 27: Skizze: Factroy 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 SerializerFactroy benutzt. Diese folgt dem Singleton Pattern und enhält 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 SerializerFactroy 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 Kontext eines Deskriptor

Für den Algorithmus der Extraktion des Kontext 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 28 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 werde 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 gerbte 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 Knote 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 28.

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 Schliefen auftauchen dürfen. Andernfalls könnte keine Wurzel gefunden werden.

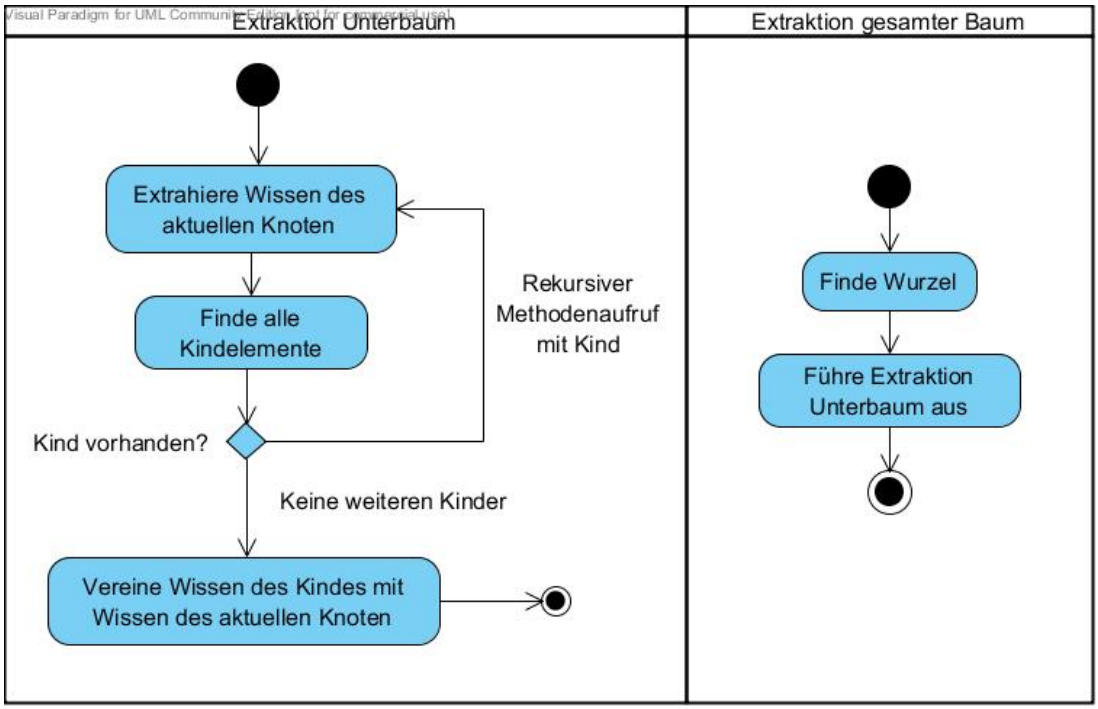

Abbildung 28: Skizze: Algorithmus Extraktion

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.

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.

TODO: Abbildung einfügen + Erklärung

5.2 Codequalität

Zur Sicherstellung der Qualität des Quellcodes wurde das Werkzeug PMD genutzt.

TODO: Abbildung einfügen + Erklärung

5.3 Umsetzung und Test der Anforderungen

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

Funktionale Anforderungen

- **Beschreibbarkeit:** Die Beschreibbarkeit ist durch den Deskriptor umgesetzt worden. Es existieren Unit Test für diesen, die Fehlerfrei durchlaufen. Des Weiteren zeigt die Chat Anwendung, dass reale Programme mit diesem Hilfemittel geschrieben werden können.
- **Beziehungen:** Die Beziehungen wurden durch das Schema umgesetzt. Auch hier zeigen Unit Tests und die Chat Anwendung die Erfüllung dieser Anforderung.
- **Persistenz:** Über das Schema können Beschreibungen gespeichert und geladen werden. Unit Test beweisen, dass diese Anforderung umgesetzt wurde.
- **Synchronisation:**
- **Änderbarkeit:** Deskriptoren besitzen einen Identifikator über den sie Unabhängig von ihres Kontextes auffindbar sind. Somit kann eine Änderung ohne Verlust des Deskriptor durchgeführt werden. Beweis hierfür sind Unit Tests des Schemas.
- **Änderung kommunizieren:**

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. Somit wurde diese Anforderung umgesetzt.
- **Testbarkeit:** Es existieren Unit Test, daher ist der Quellcode testbar.
- **Modultest:** Es wurden eine Reihe von Unit Test geschrieben und ein Cobertura Report erstellt. Somit ist diese Anforderung erfüllt.
- **Wartbarkeit:** Der PMD Report erzeugt nur unwesentliche Meldungen, die ignoriert werden können (sie 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 genutzt werden.

6 Fazit

7 Quellenverzeichnis

Artikel

[1] 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).

[2] 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

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

Handbücher

[4] 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

[5] Scott A. Golder und Bernardo A. Huberman. *The Structure of Collaborative Tagging Systems*.

[6] Peter Pin-Shan Chen. *The Entity-Relationship Model-Toward a Unified View of Data*. 1976.

[7] Randall Reynolds Clemente Izurieta Nurzhan Nurseitov Michael Paulson. *Comparison of JSON and XML Data Interchange Formats: A Case Study*.

[8] Mor Naaman und Chen Ye Oded Nov. *What Drives Content Tagging: The Case of Photos on Flickr*. 2008.

Spezifikationen

[9] 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

[10] *GitHub*. URL: https://de.wikipedia.org/wiki/Dateisystem (besucht am 30. 09. 2015).

[11] *Github SharkFW - SyncKB*. URL: https://github.com/SharedKnowledge/SharkFW/tree/master/src/java/core/net/sharkfw/knowledgeBase/sync (besucht am 21. 09. 2015).

[12] Stefan Hagen. *Dateisystem - Wikipedia*. URL: https://de.wikipedia.org/wiki/Dateisystem (besucht am 30. 09. 2015).

[13] *JavaDoc - XmlAdapter*. URL: https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html (besucht am 04. 10. 2015).

[14] *Jenkins*. URL: http://jenkins-ci.org/ (besucht am 15. 09. 2015).

[15] *opendatastructures.org - 12. Graphs*. URL: http://opendatastructures.org/versions/edition-0.1g/ods-python/12_Graphs.html (besucht am 28. 09. 2015).

[16] *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).

[17] *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).

[18] Prof. Dr. Thomas Schwotzer. *Shark framework.* URL: http://www.sharksystem.net/ (besucht am 31. 08. 2015).

[19] *Skype.* URL: http://www.skype.com/de/home/ (besucht am 30. 09. 2015).

[20] *stackoverflow.* URL: http://stackoverflow.com/questions/32942615/how-to-convert-dot-separated-string-into-array-list (besucht am 05. 10. 2015).

[21] Paul E. Black und. *Dictionary of Algorithms and Data Structures.* URL: http://xlinux.nist.gov/dads//HTML/tree.html (besucht am 26. 09. 2015).

[22] *WoltLab GmbH - Burning Board Community Forum.* URL: https://community.woltlab.com/ (besucht am 30. 09. 2015).

8 Abbildungsverzeichnis

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: