



Masterarbeit

Ein System zur partiellen Synchronisation von Wissensbasen für dezentrale soziale Netzwerke

von Jens Grundmann
4. Oktober 2015

Hochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

Erstgutachter/in Prof. Vorname Name
Zweitgutachter/in Vorname Name

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung | 2 |
| 2 | Grundlagen | 3 |
| 2.1 | Funktionale und nicht funktionale Anforderungen | 3 |
| 2.2 | Shark Framework | 3 |
| 2.2.1 | Context Space | 4 |
| 2.2.2 | Knowledge Base | 5 |
| 2.2.3 | Knowledge Port | 5 |
| 2.2.4 | SyncKB | 6 |
| 2.3 | Datenstrukturen zur Darstellung von Beziehungen | 8 |
| 2.3.1 | Verkettete Listen | 8 |
| 2.3.2 | Bäume | 9 |
| 2.3.3 | Graphen | 9 |
| 2.3.4 | Entity Relationship Modell | 10 |
| 2.4 | Aufbau von Social Media Formaten | 10 |
| 2.4.1 | Chat | 10 |
| 2.4.2 | Forum | 11 |
| 2.4.3 | Dateisysteme und Versionsverwaltung Software | 13 |
| 3 | Konzeption | 14 |
| 3.1 | Serialisierung | 14 |
| 3.2 | Unterscheidung in Beschreibung und Synchronisation | 17 |
| 3.3 | Darstellung eines Datenbereiches: Der Deskriptor | 17 |
| 3.3.1 | Analyse der Datenstrukturen | 17 |
| 3.3.2 | Beschreibung eines Datenbereiches durch einen Deskriptoren | 18 |
| 3.3.3 | Gleichheit von Deskriptoren | 19 |
| 3.3.4 | Darstellung und Bedeutung von Beziehungen | 19 |
| 3.3.5 | Ein Schema von Beziehungen | 22 |
| 3.3.6 | Extraktion von Daten | 23 |
| 3.4 | Synchronisation | 23 |
| 3.4.1 | Mängel der aktuellen SyncKB | 23 |
| 3.4.2 | Abstraktion von der SyncKB | 24 |
| 4 | Implementierung | 25 |
| 4.1 | Deskriptor und Schema | 25 |
| 4.2 | Serialisierung von Deskriptor und Schema | 26 |
| 5 | Quellenverzeichnis | 27 |
| 6 | Abbildungsverzeichnis | 28 |
| A | CD-ROM zur Arbeit | 29 |
| B | Eigenständigkeitserklärung | 30 |

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 bedeutet, 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 der Benutzer. Das Client-Server Modell wird durch ein Peer to Peer Model ersetzt. Daten werden nur mit den Personen ausgetauscht, für die sie bestimmt sind. Hier ist eine Methode benötigt, 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 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 [14] von Prof. Dr. Thomas Schwotzer und die darin enthaltene SyncKB Klassensammlung. Diese ermöglicht bereits eine Synchronisation aller Daten in einer Wissensbasis. Diese soll nun dahingehend ausgebaut werden, dass Teile der Wissensbasis beschrieben werden können und nur diese beschriebenen Teile synchronisiert werden.

Zuerst muss mit einer Möglichkeit gefunden werden, wie ein Teilbereich der Wissensbasis beschrieben werden kann. Diese 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 der 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 der Funktionalität der Softwarekomponente wird schließlich eine Chat mit grafischer Oberfläche geschrieben, in dem einzelne Peers sich miteinander unterhalten können.

Die Arbeit wird zuerst die Grundlagen, wie das Shark Framework [14], auf denen die Implementierung beruht, erläutern. Danach wird das Konzept der Softwarekomponente erstellt. Es werden verschiedene Möglichkeiten 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 zur 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 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 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 werden können wie zuvor.
- **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 [10], sind wünschenswert.
- **Testbarkeit:** Die zu Softwarekomponente ist modular so aufgebaut, dass sie durch Modultest testbar ist.
- **Modultest:** Es existieren bereits eine Reihe von Modultest, welche die grundlegende Funktionalität der Softwarekomponente sicherstellen.
- **Wartbarkeit:** Der Code der Software soll wartbar sein. Dies bedeutet der muss verständlich geschrieben sein und Fehler müsse möglichst einfach aufspürbar sein.

2.2 Shark Framework

In diesem Unterkapitel wird auf die grundlegenden Features des Shark Framework [14] 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 die Context Space, der hier vereinfacht Kontext genannt werden soll. Hierbei handelt es sich um eine Datenstruktur. Abbildung 1 zeigt ein vereinfachtes Modell dieser Struktur.

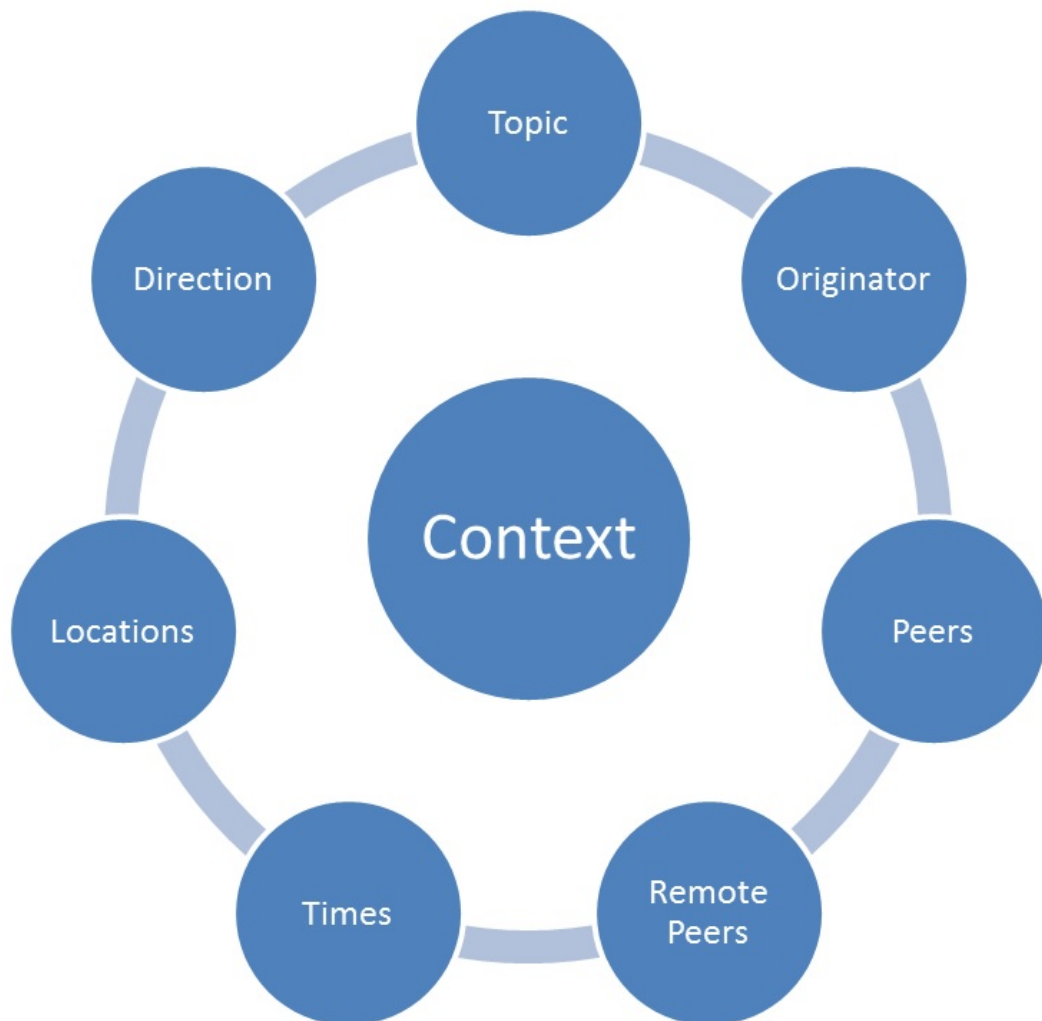


Abbildung 1: Shark Context Space Modell

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 der beschriebenen Kontextes. Die kann entweder der Ersteller der sein oder Autor der beiliegenden Informationen. Bei einem wissenschaftlichen Artikel wäre dies der Autor des Titels.
- **Peers:** Peers kann unterschiedlich interpretiert werden. Zum einen kann es der Ersteller des Kontextes sein, der sich vom Autor der beiliegenden Informationen unterscheiden kann. Zum anderen kann es der Besitzer des eines Kontextes sein, wobei Besitzer hier je nach beiliegendem Fall anders interpretiert werden kann. Im Allgemeinen kommt die Interpretation dieser 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. Diese Information kann je nach vorliegendem Fall anders interpretiert werden.
- **Direction:** Die Richtung der Kommunikation des Kontextes. Es ist möglich nur Informationen über einen Kontext zu empfangen, sie nur zu senden, zu lesen und senden und keine der genannten Aktionen durchzuführen.

Zu beachten 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. Währenddessen besitzt der Kontextpunkt nur je genau einen Inhalt. Beispielfhaft beschreibt der Kontextpunkt nur das Thema Java, während der Kontext Java und C++ beinhaltet. 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 bildet 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 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 für, 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 kann mithilfe dieses Objektes Daten an die *doExpose* und *doInsert*-Methode anderer Peer gesendet werden. Dies erlaubt einen mehrfachen Datenaustausch. Ein Beispiel hierzu ist der Synchronisationsprozess der SyncKB, der im nächsten 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 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 Funktionsweise der SyncKB. Daher soll hier auf die generelle Funktionsweise dieser eingegangen werden. Weitere Informationen können den Quellcode selbst entnommen werden, der im Github vom SharkFW zu finden ist. [8]

Die SyncKB enthält 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 1 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.

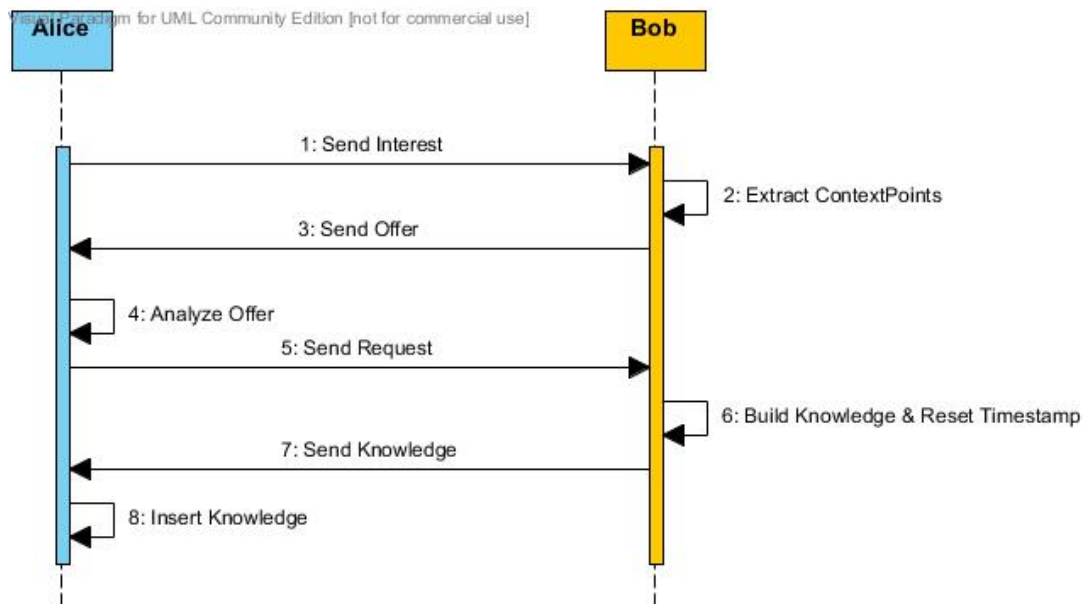


Abbildung 2: Kommunikation der SyncKB

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 Alice als Angebot zugewendet. Zu beachten ist dabei, dass nur die Daten eines Kontextpunktes versendet werden. Eventuelle Informationen, die diesem zugeordnet sind, werden nicht versandt.
4. **Angebot analysieren:** Alice analysiert das Angebot, dass sie von Bob erhalten hat. Der Algorithmus ist ähnlich dem späteren Einfügen von 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 zurücksetzen:** Die angefragten Kontextpunkte werden nun aus Bobs Wissensbasis extrahiert und in einem Knowledge Objekt gespeichert. Zusätzlich werden alle Themen und Peers, anhand der beim Erstellen des SyncKP übergebenen FragmentationParameter, dem Knowledge Objekt übergeben.
7. **Wissen senden:** Das Knowledge Objekt wird nun an Alice gesendet.
8. **Wissen in Wissensbasis einfügen:** Alice überprüft das die Kontextpunkte anschließend noch einmal anhand das im rechten Teil von Abbildung 3 skizzierten Algorithmus und fügt diese dann in ihre Wissensbasis ein.

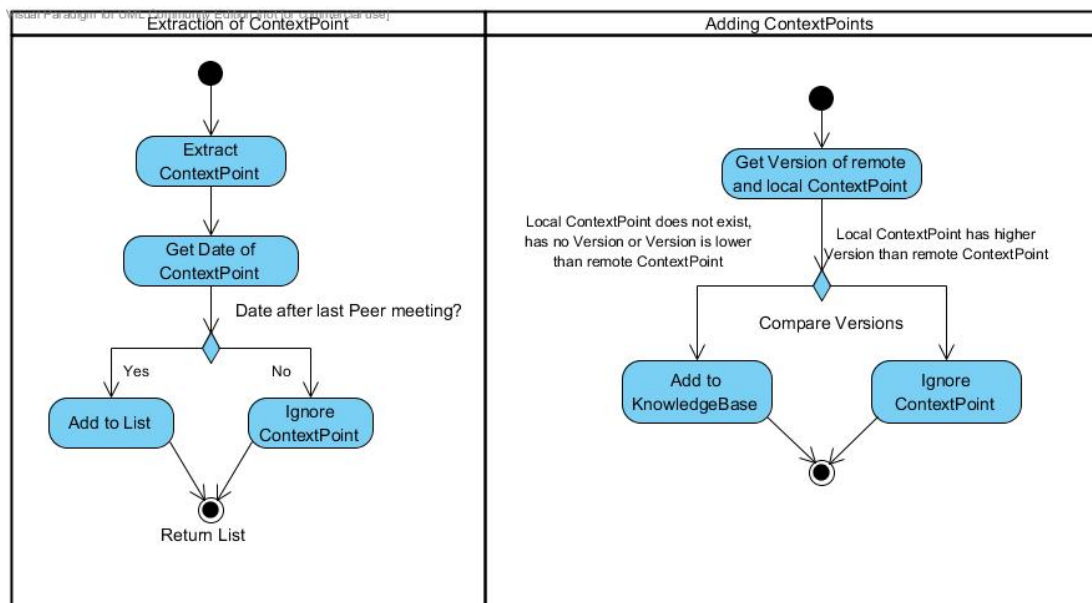


Abbildung 3: Algorithmus zum Extrahieren und Einfügen von Wissen der SyncKB

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 zusätzlich 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 Verkettete Listen

Verkettete Listen (beschrieben in [3], Kapitel 4) sind Listen, wo jedes Element Referenzen auf weitere Mitglieder der Liste enthält. Hier sollen die Einfach-Verketteten-Listen und die Zweifach-Verketteten-Listen betrachten 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 [12] entnommen werden.

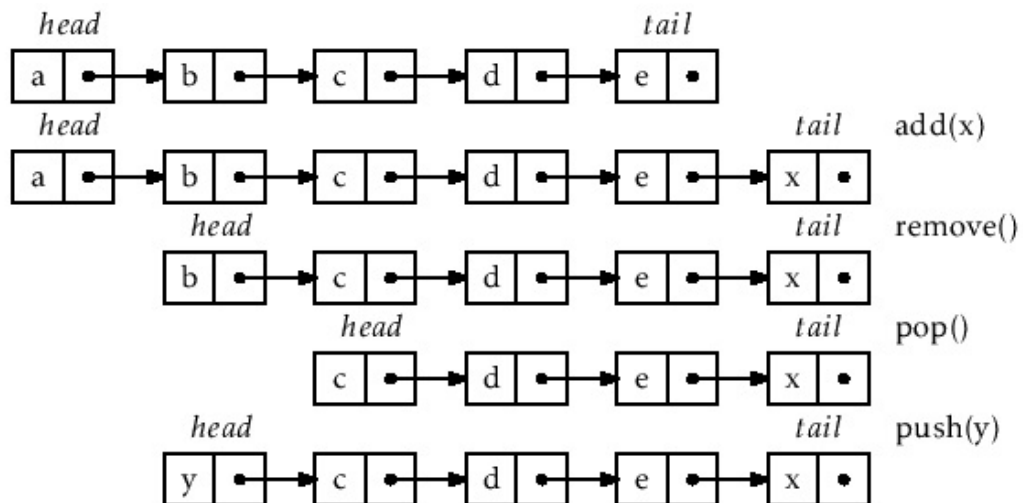


Abbildung 4: Aufbau und Operation einer einfach verketteten Liste. Quelle: [12]

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 [13] entnommen werden.

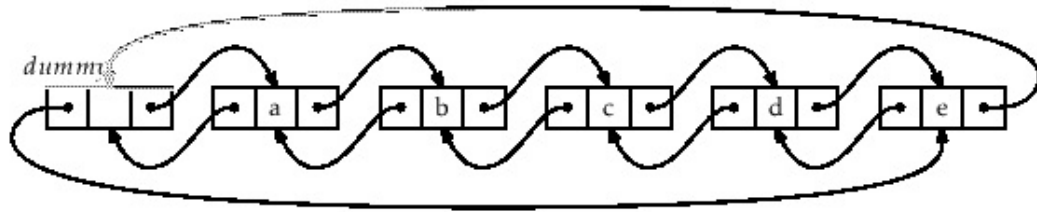


Abbildung 5: Aufbau einer zweifach verketteten Liste. Quelle: [13]

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 besondere Eigenschaft der ist, dass sie keinen Vater besitzt. Knoten, die keine Kinder besitzen, werden Blätter genannt. Ein Element kann hier also 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.

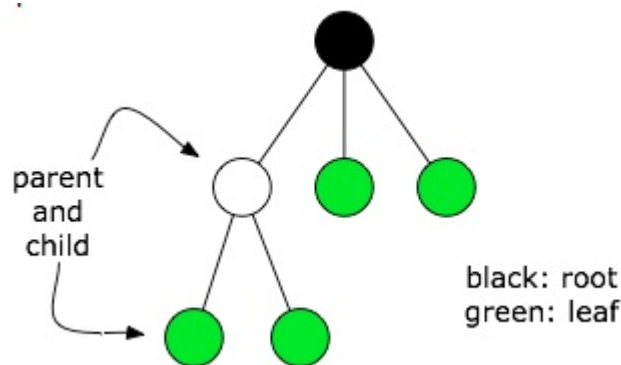


Abbildung 6: Aufbau eines Baums. Quelle: [16]

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, der den Anfang darstellt. Zudem sind die Beziehungen nicht fest gelegt. Ein Element besitzt eine Vielzahl an Beziehungen. Gerichtet 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 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 unbeschrieben. Abbildung 7 skizziert das Modell eines gerichteten Graphen. Die Peile stellen dabei die Übergänge zwischen den Elementen da. Weitere Erklärungen könne unter [11] gefunden werden.

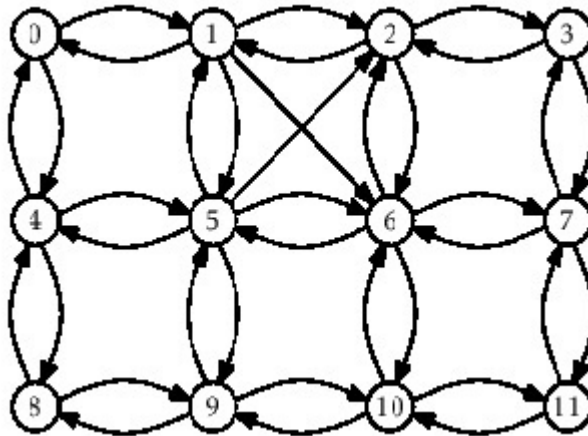


Abbildung 7: Aufbau eines Graphen. Quelle: [11]

2.3.4 Entity Relationship Modell

Das Entity Relationship Modell, beschrieben in [5], ist weniger ein 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 jede 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 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.

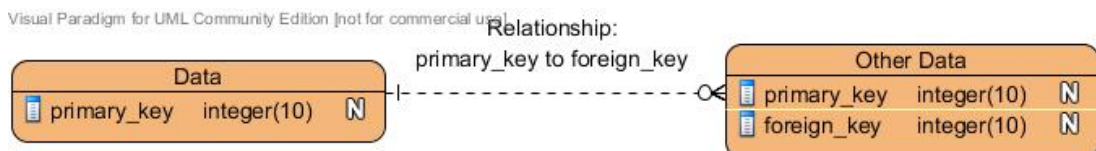


Abbildung 8: Vereinfachte Skizze des Entity Relationship Modell

2.4 Aufbau von Social Media Formaten

In diesem Abschnitt soll der Aufbau 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.

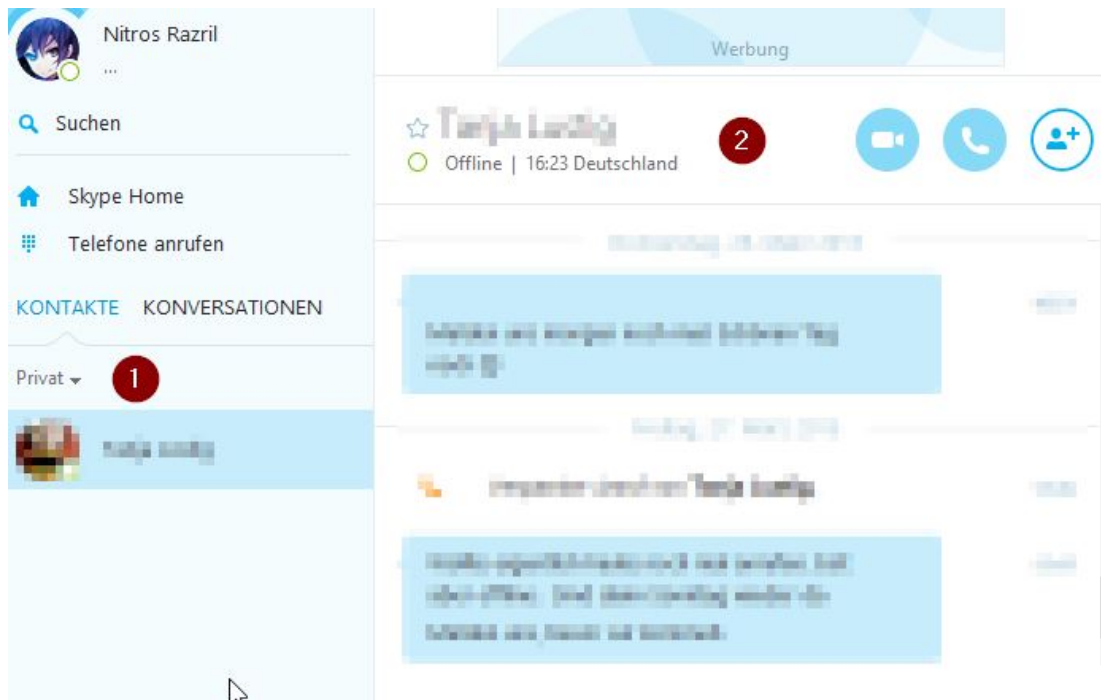


Abbildung 9: Aufbau Chat in Skype. [15]

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 beschrieben sind.

2.4.2 Forum

Ein Forum ist gegenüber den Chat meist komplexer und lässt sich in mehr Ebenen einteilen. Als Beispiel soll das Burning Board der WoltLab GmbH herangezogen werden [17]. In den folgende Abbildungen 10, 11 und 12 ist der strukturelle Aufbau eines Forums zu sehen.



Abbildung 10: Forumstruktur: Oberste Ebene. Quelle: [17]

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.

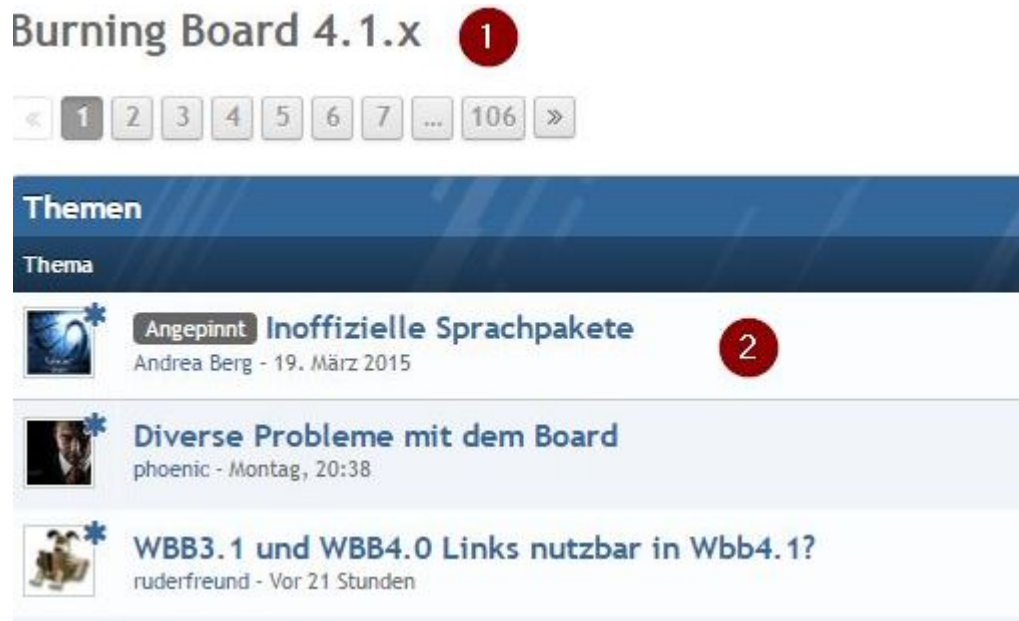


Abbildung 11: Forumstruktur: Thread-Sammlung Ebene. Quelle: [17]

?? 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.

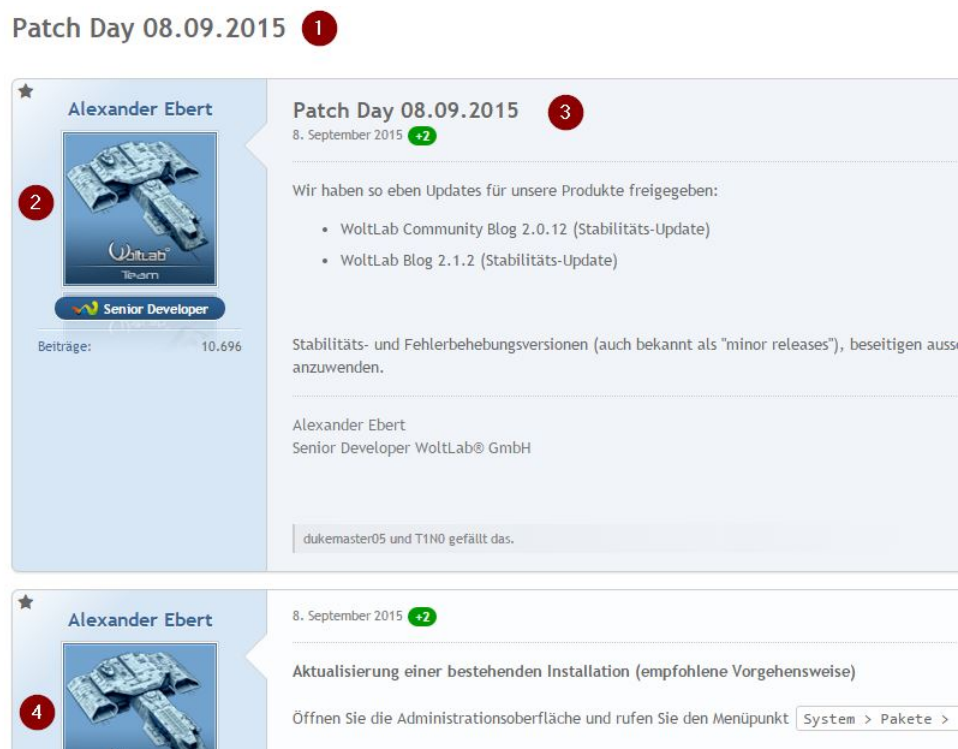


Abbildung 12: Forumstruktur: Thread Ebene. Quelle: [17]

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 genauso ähnliche Daten, wie zum Beispiel 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 nur die Blätter Daten enthalten. Andere Knoten dienen lediglich einer Ordnung der Daten und der Eingruppierung dieser in einen definierten Bereich.

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 [2] wird beschrieben, dass man immer mehr auf den webbasierten Filehosting-Dienst für Software-Entwicklungsprojekte GitHub [7] stößt, der auf Git basiert. Mit seiner Vielzahl an Projekten der unterschiedlichsten Programmiersprachen kann GitHub als soziale Plattform für Softwareentwickler gesehen werden.

Aus diesem soll der Austausch von Teilen des Software über solche Versionsverwaltung Software auch Betrachtung finden. Abbildung 13 zeigt den Allgemeinen von Dateisystem für unterschiedliche Betriebssysteme.

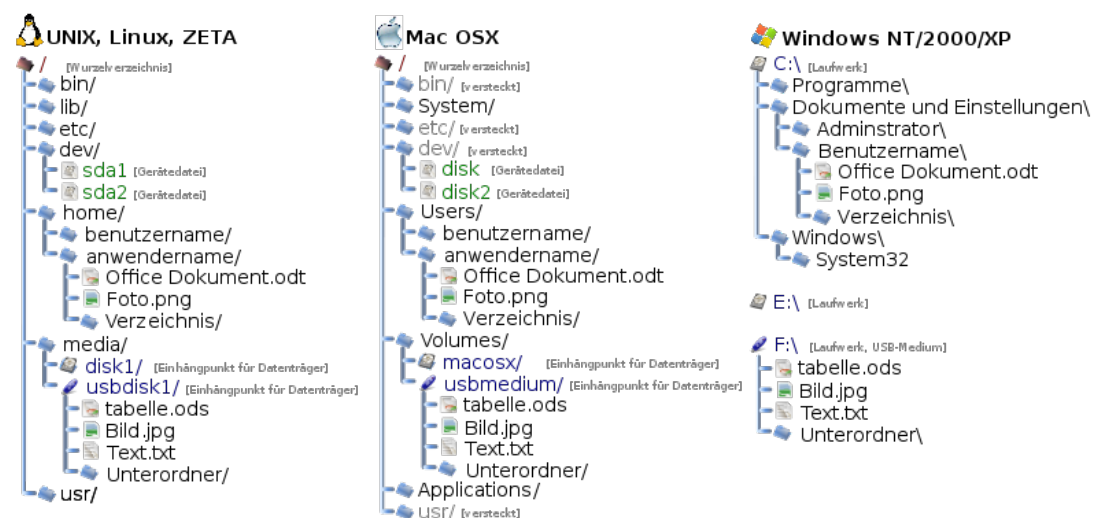


Abbildung 13: Illustration über den Vergleich von Dateisystem-Bäumen. Quelle: [9]

Zu sehen ist, dass identisch zum Forum eine Baumstruktur die Grundlage von Dateisystemen ist. Ebenfalls enthalten auch hier nur die Blätter die die eigentlichen Daten und die anderen Knoten dienen 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 beschreibbaren Raum von Daten bestand.

3 Konzeption

In diesem Abschnitt wird die Konzeption der geplanten Softwarekomponente besprochen. Basierend 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 [6] zu rate gezogen. Diese 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 14: Scenario 1 JSON vs. XML Timing. Quelle: [6]

| | 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 15: Scenario 1 JSON vs. XML CPU/Mem. Quelle: [6]

Die Abbildungen 14 und 15 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 16: Scenario 2 JSON Vs XML Timing. Quelle: [6]

| 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 17: Scenario 2 JSON CPU/Mem. Quelle: [6]

| 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 18: Scenario 2 XML CPU/Mem. Quelle: [6]

Die Abbildungen 16, 17 und 18 zeigen den Zweiten Fall der Fallstudie. Hier wurden mehrfach hintereinander kleine Datensätze übertagen 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". Dies beschreibt, dass Frameworks ihre Abhängigkeiten, in der Form von Jar-Archiven, mitbringen und dadurch die Möglichkeit besteht diese mehrfach in einem Projekt zu haben. Dies 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 diese 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 dies kann durchaus verwirrend für Entwickler sein. Besonders, im Bezug auf die Weiterentwicklung des Frameworks. Die zur Zeit prioritäre XML Serialisierung sollte einfacher auf das in Java Standard Edition enthaltene JAXB umzustellen sein als auf ein externes Framework für JSON. Sofern dies in bei zukünftigen Refactoring Aktionen geschieht.
- **XML Schema beschrieben:** 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 gewinnen könnte. Speziell 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 stellt das dem Extrahieren von Daten. Wie mit diesen Daten umgegangen wird ist nicht festgelegt. Währenddessen einhält die Synchronisation die eigentliche Logik zum Ausführen der partiellen Synchronisation.

3.3 Darstellung eines Datenbereiches: Der Deskriptor

In diesem Abschnitt wird besprochen, wie ein Datenbereiches 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, sodass 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.

3.3.1 Analyse der Datenstrukturen

Im folgenden die Analyse der in Abschnitt 2.3 beschriebenen 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-Verkettete-Listen könnte hier vorwärts und rückwärts nach Element gesucht werden. Serialisiert wären sie in XML durch eine einfache Liste mit einer vorgegebenen 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 Teils des Dateisystems.

Bäume

Die meisten der besprochenen Social Media Formate haben von Grund 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 Datenstruktur hier enthält.

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 seinem Kindern kommt und das Kind immer seinen Vater kennt. Bäume sind für diese ausreichend, besonders da diese einen Startpunkt, die Wurzel, besitzen.

Entity Relationship Modell

Auch wenn eher ein Modell als eine Dateistruktur, 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 reichen. Dies ermöglicht die Flexibilität, dass ein Element sehr leicht mehrere Kinder und Väter haben. Auch als 1:1, 1:N, N:M Beziehung aus Datenbanksystemen bekommt. 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 Fremdschlüssels ebenfalls einfach verwiesen werden. Beziehungen daher, entsprechen des Namens des Modells, leicht abzubilden. Wenn es 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 zusammen gesetzt werden müssen. Wir haben im Rahmen dieser Arbeit keine relationale Datenbank mit einem Datenbanksystem, welche das für uns 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. Serialisiert kann das ganze so als eine Liste werden, wobei jeder Identifikator nur einmal vorkommt.

3.3.2 Beschreibung eines Datenbereiches durch einen Deskriptoren

Nach der Analyse aus dem vorhergehenden Abschnitt soll nun festgelegt werden mit welchen Parametern die ein Datenbereiches beschrieben wird. Abbildung 19 zeigt den Aufbau unseres beschreibenden Elements, dem Deskriptor.

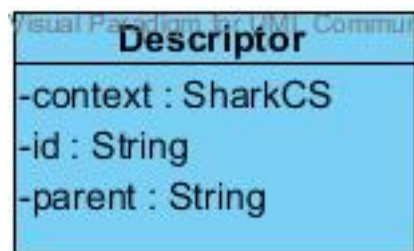


Abbildung 19: Konzeption eines Deskriptor

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 ist möglich, wenn der Deskriptor 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 sein, 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 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.

3.3.3 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 übereinstimmen.
- **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 übereinstimmen. Des Weiteren muss der Kontext, nach Regeln des Shark Frameworks, identisch sein.

Diese Unterscheidung ist notwendig, um das wiederzufinden eines Deskriptor sicher zu stellen und doppelte Datenhalten zu verhindern, da der Identifikator einen Primärschlüssel darstellt. Anhand dessen kann nach einem bestimmten Deskriptor gesucht werden. dennoch muss es möglich sein den Unterschied in Vater und Kontext zu erkennen für mögliche Anpassungen. Zum Beispiel könnte der Deskriptor von einem anderen Peer geändert worden sein und wurde dann versendet, 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. Ob ein Deskriptor zu einem Anderen identisch ist soll anhand einer gesonderten Methode implementiert werden.

3.3.4 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 20 dargestellt ist.

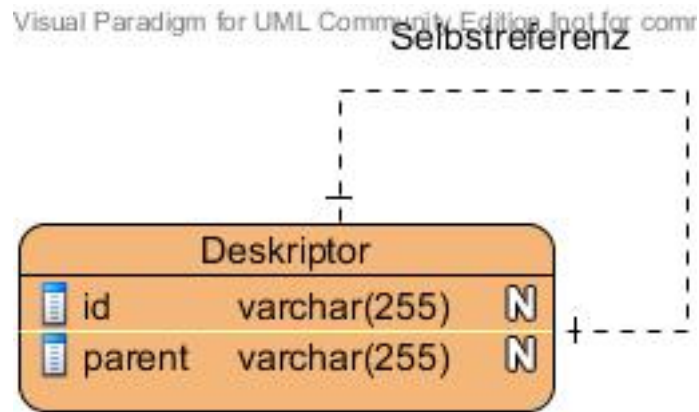


Abbildung 20: Skizze: Selbstreferenz

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 beliebig viele Kinder haben, währendes ein Kind nur einen Vater besitzen kann.

Folgendes soll die Anwendung dieses Modells 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 21 zeigt ein Konzept, wie für Beziehungen von Chats.

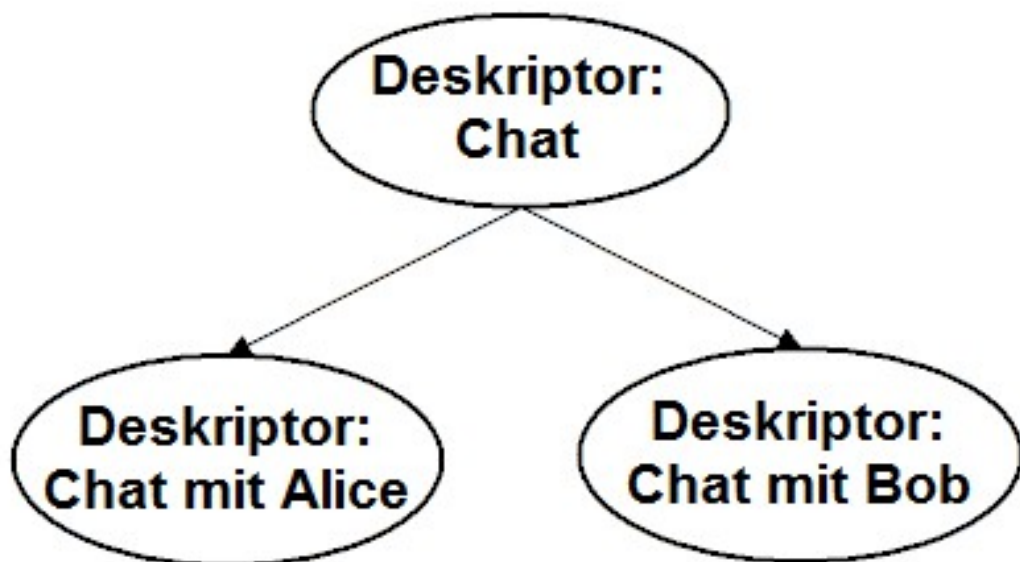


Abbildung 21: Beziehungen von Deskriptoren für Chats

Hier existiert genau eine Vater-Kind Beziehung. Dabei ist die Wurzeln des Baumes nur ein

Anker, der andere Deskriptoren zusammenfasst. Geht man nach Abbildung 21, so ist es möglich alle Chats zu finden, indem man alle Kinder des Deskriptor Chat 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 sein kann. Auf der anderen Seite gibt es die Elemente, welche die eigentlichen Daten enthalten.

Abbildung 22 zeigt ein Konzept für Beziehungen, welche für die Einordnung von Unterforen und Threads in ein Forum darstellt.

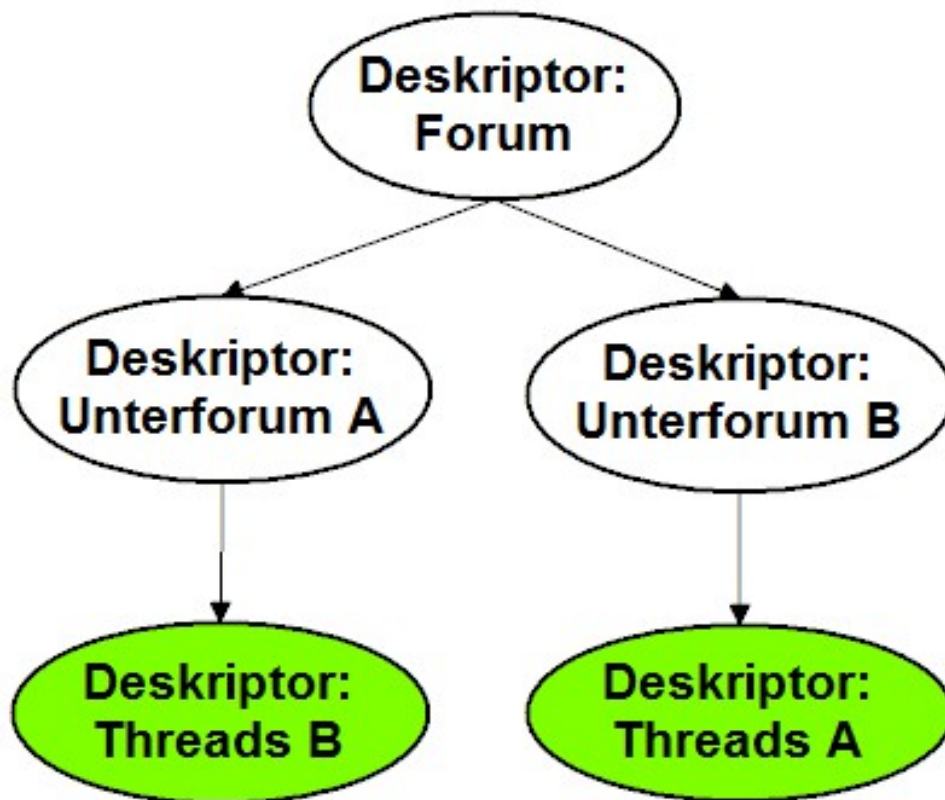


Abbildung 22: Beziehungen von Deskriptoren eines Forum

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 zur Darstellung der Beziehung verwendet werden. Zumindest ein Blatt muss 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 einen Thread. Ein nicht leerer Deskriptor mit einer Beziehung sollte

Aufgrund der Übersichtlichkeit nicht verwendet werden.

Die Struktur von Dateisystem ist gleich der eines Forum. Abbildung 23 zeigt die Beziehungen für Verzeichnisse und Dateien.

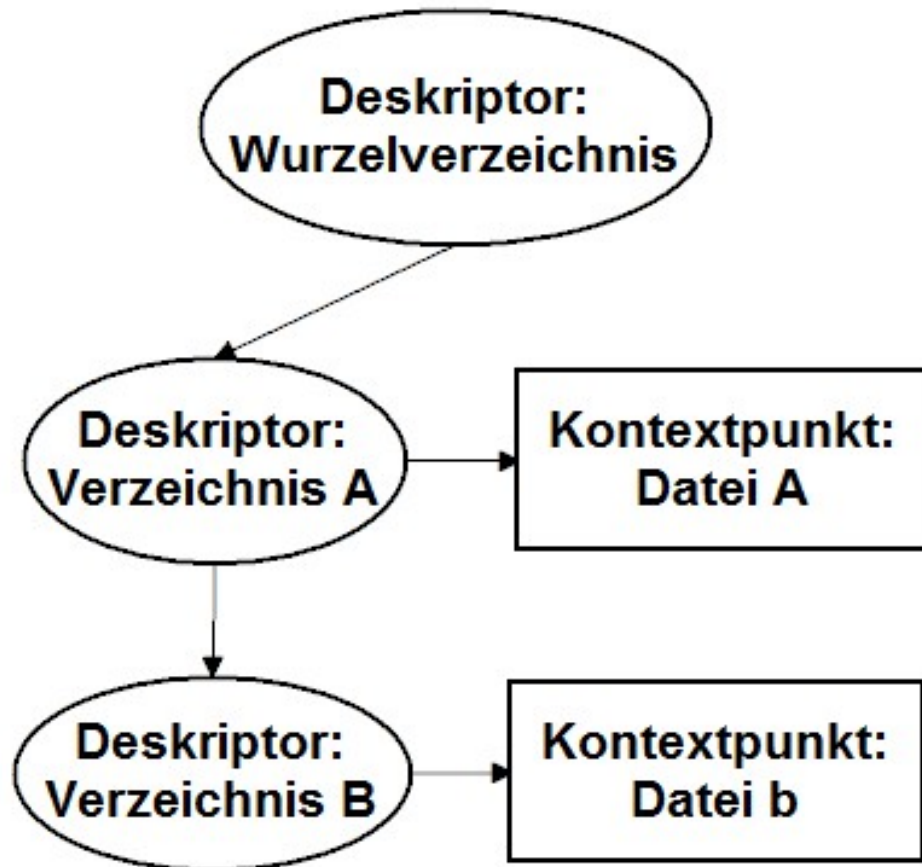


Abbildung 23: Beziehungen von Deskriptoren in einem Dateisystem

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 beachte, dass es sich hierbei um ein Konzept handelt. Implementieren 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.

3.3.5 Ein Schema von Beziehungen

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.

3.3.6 Extraktion von Daten

Deskriptoren existieren um Datenbereiche zu beschreiben. Als solches ist es 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.

3.4 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.

3.4.1 Mängel der aktuellen SyncKB

Bevor ein Algorithmus entworfen wird sollen die Mängel der aktuellen Synchronisation besprochen werden. Abbildung 24 zeigt nochmal eine Skizze des Algorithmus des SyncKP aus Abschnitt 2.2.4. Für den eine Erklärung des sequenziellen Ablauf siehe Abbildung 2 und Nachfolgende Beschreibung.

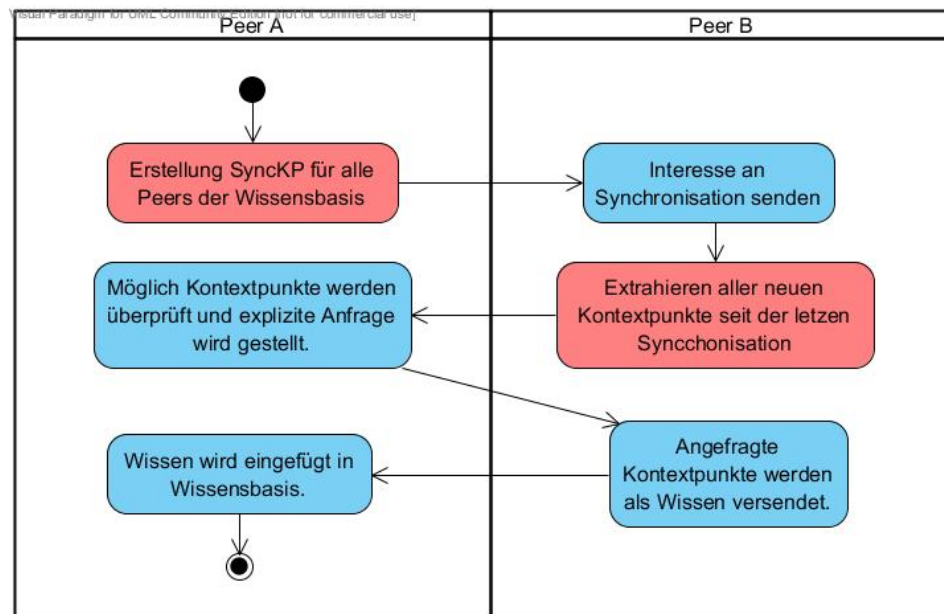


Abbildung 24: Skizze: Synchronisation Algorithmus der aktuellen SyncKB

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 Beiepeilweise 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 Name ist allerdings eine partiellen Synchronisation erwünscht.

3.4.2 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 ist hier von einer Abstraktion in der objektorientierte Programmierung die Rede. Die Klasse SyncKP soll in eine abstrakte Klasse AbstractSyncKP umgewandelt werden. Der Entwickler, der von dieser erbt ist dann, ist dann verpflichtet die abstrakten Methoden dieser Klasse zu implementieren. Ziel dieser ist das Erreichen folgender Aktionen:

- **Entscheidung ob ein Interesse besteht:** Es sollte im Ermessen des Entwicklers liegen, ob eine Interesse zur Synchronisation 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 3.3.6 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 zu extrahieren 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

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 25 zeigt das Klassendiagramm beider.

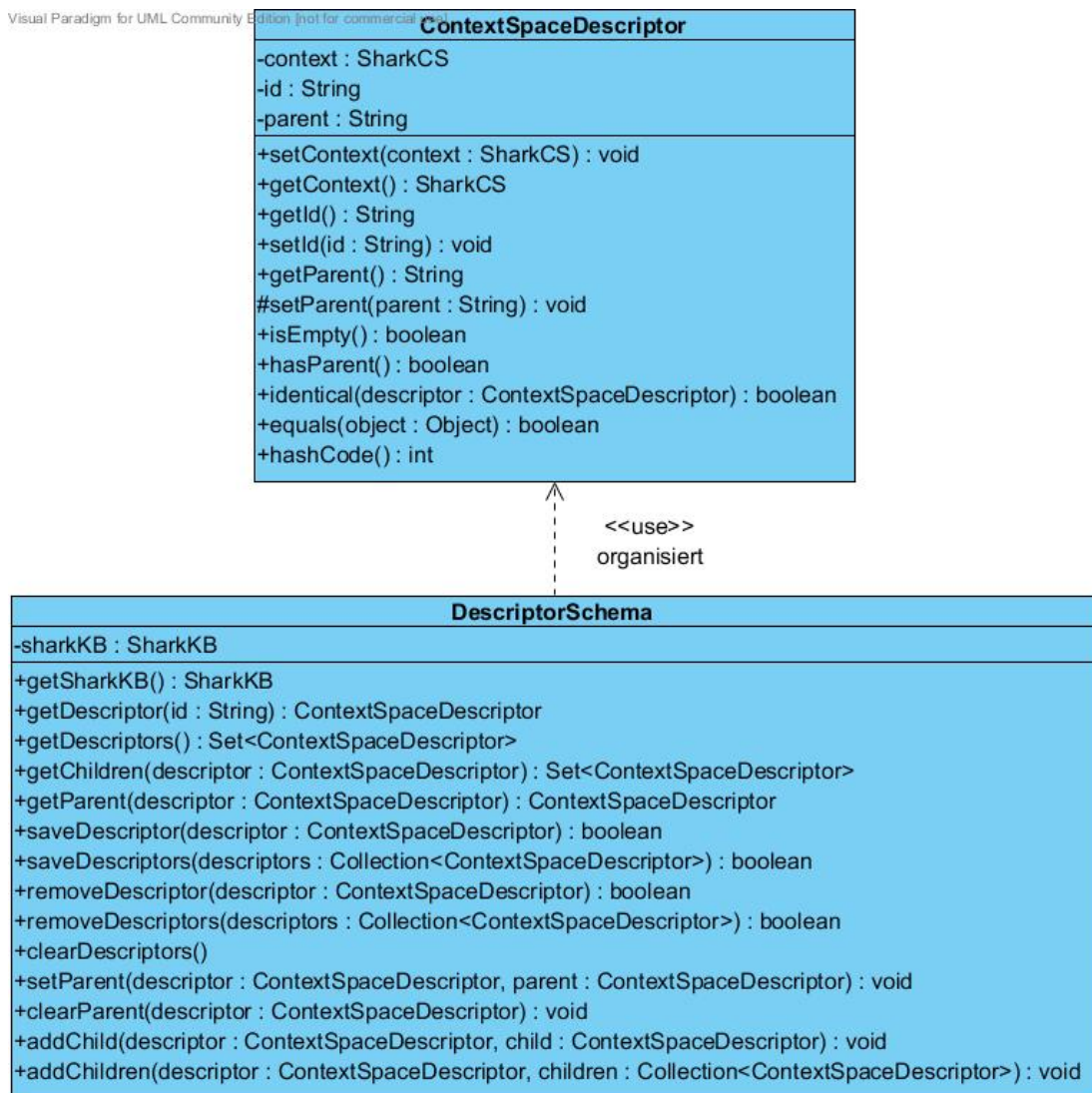


Abbildung 25: Klassendiagramm: Deskriptor und Schema

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

Schema

4.2 Serialisierung von Deskriptor und Schema

Wie in Abschnitt 3.1

5 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] Peter Pin-Shan Chen. *The Entity-Relationship Model-Toward a Unified View of Data*. 1976.
- [6] Randall Reynolds Clemente Izurieta Nurzhan Nurseitov Michael Paulson. *Comparison of JSON and XML Data Interchange Formats: A Case Study*.

Webseiten

- [7] *GitHub*. URL: <https://de.wikipedia.org/wiki/Dateisystem> (besucht am 30.09.2015).
- [8] *Github SharkFW - SyncKB*. URL: <https://github.com/SharedKnowledge/SharkFW/tree/master/src/java/core/net/sharkfw/knowledgeBase/sync> (besucht am 21.09.2015).
- [9] Stefan Hagen. *Dateisystem - Wikipedia*. URL: <https://de.wikipedia.org/wiki/Dateisystem> (besucht am 30.09.2015).
- [10] *Jenkins*. URL: <http://jenkins-ci.org/> (besucht am 15.09.2015).
- [11] *opendatastructures.org - 12. Graphs*. URL: http://opendatastructures.org/versions/edition-0.1g/ods-python/12_Graphs.html (besucht am 28.09.2015).
- [12] *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).
- [13] *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).
- [14] Prof. Dr. Thomas Schwotzer. *Shark framework*. URL: <http://www.sharksystem.net/> (besucht am 31.08.2015).
- [15] *Skype*. URL: <http://www.skype.com/de/home/> (besucht am 30.09.2015).
- [16] Paul E. Black und. *Dictionary of Algorithms and Data Structures*. URL: <http://xlinux.nist.gov/dads/HTML/tree.html> (besucht am 26.09.2015).
- [17] *WoltLab GmbH - Burning Board Community Forum*. URL: <https://community.woltlab.com/> (besucht am 30.09.2015).

6 Abbildungsverzeichnis

| | | |
|----|---|----|
| 1 | Shark Context Space Modell | 4 |
| 2 | Kommunikation der SyncKB | 6 |
| 3 | Algorithmus zum Extrahieren und Einfügen von Wissen der SyncKB | 7 |
| 4 | Aufbau und Operation einer einfach verketteten Liste. Quelle: [12] | 8 |
| 5 | Aufbau einer zweifach verketteten Liste. Quelle: [13] | 9 |
| 6 | Aufbau eines Baums. Quelle: [16] | 9 |
| 7 | Aufbau eines Graphen. Quelle: [11] | 10 |
| 8 | Vereinfachte Skizze des Entity Relationship Modell | 10 |
| 9 | Aufbau Chat in Skype. [15] | 11 |
| 10 | Forumstruktur: Oberste Ebene. Quelle: [17] | 11 |
| 11 | Forumstruktur: Thread-Sammlung Ebene. Quelle: [17] | 12 |
| 12 | Forumstruktur: Thread Ebene. Quelle: [17] | 12 |
| 13 | Illustration über den Vergleich von Dateisystem-Bäumen. Quelle: [9] | 13 |
| 14 | Scenario 1 JSON vs. XML Timing. Quelle: [6] | 14 |
| 15 | Scenario 1 JSON vs. XML CPU/Mem. Quelle: [6] | 14 |
| 16 | Scenario 2 JSON Vs XML Timing. Quelle: [6] | 15 |
| 17 | Scenario 2 JSON CPU/Mem. Quelle: [6] | 15 |
| 18 | Scenario 2 XML CPU/Mem. Quelle: [6] | 16 |
| 19 | Konzeption eines Deskriptor | 18 |
| 20 | Skizze: Selbstreferenz | 20 |
| 21 | Beziehungen von Deskriptoren für Chats | 20 |
| 22 | Beziehungen von Deskriptoren eines Forum | 21 |
| 23 | Beziehungen von Deskriptoren in einem Dateisystem | 22 |
| 24 | Skizze: Synchronisation Algorithmus der aktuellen SyncKB | 23 |
| 25 | Klassendiagramm: Deskriptor und Schema | 25 |

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: