Improving Data Quality using Domain-Specific Data Types

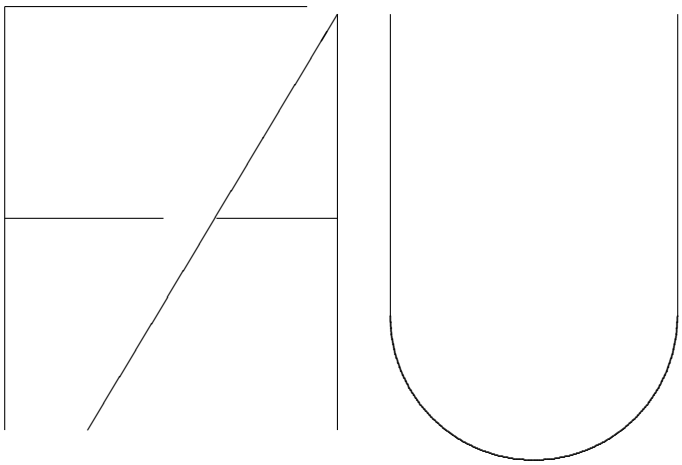
Masterarbeit

Patrick Reischl

Professur für
Open-Source-Software

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



Improving Data Quality using Domain-Specific Data Types

Masterarbeit im Fach Informatik

vorgelegt von

Patrick Reischl

geb. 01.05.1989 in Nürnberg

angefertigt am

Department Informatik
Professur für Open-Source-Software
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.

Beginn der Arbeit: 01.04.2014
Abgabe der Arbeit: 30.09.2014

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nürnberg, den 30.09.2014

(Patrick Reischl)

License

Nürnberg, den 30.09.2014

(Patrick Reischl)

Abstract

Improving Data Quality using Domain-Specific Data Types

In this thesis the concept and implementation of the Open Data Service (ODS) is described. The structure of the ODS is divided into two main components. The Open Data Service Import Component collects data from the Internet or from a local network, converts them into a JSON-serializable format if necessary, and then stores the records in the document-oriented database CouchDB. Subsequently, clients may retrieve documents by using the RESTful API of the Open Data Service Server Component. These requests are mapped onto database queries in order to provide the proper information. The ODS sample application collects free geographic data from OpenStreetMap as well as free water data like water levels and temperatures.

In addition to that, this thesis focuses on improving the data quality of the Open Data Service and Web Services in general. Therefore, a number of reusable quality improvement filters are created, which may execute simple operations on documents. Furthermore, this thesis introduces several domain-specific data types in order to describe water data. These data types are developed based on the Value Object Pattern.

Zusammenfassung

Verbesserung der Datenqualität mithilfe von domänenspezifischen Datentypen

Im Rahmen dieser Arbeit wird das das Konzept und die Implementierung des Open Data Service (ODS) vorgestellt. Die Struktur des ODS besteht aus zwei voneinander unabhängigen Hauptkomponenten. Die Importkomponente sammelt Daten im Internet oder in einem lokalen Netzwerk und konvertiert diese falls notwendig in ein JSON-serialisierbares Objektformat. Danach werden die Datensätze in einer Instanz der dokumentenorientierten Datenbank CouchDB abgelegt. Clients können auf die Daten schließlich mithilfe der, auf dem Paradigma REST basierenden, Schnittstelle der Serverkomponente zugreifen. Innerhalb dieser Komponente werden die Anfragen auf vordefinierte Datenbankabfragen übersetzt und durch diese bearbeitet. Die ihm Laufe dieser Arbeit entwickelte Beispielanwendung sammelt zum einen grundlegende geografische Daten von OpenStreetMap. Zum anderen werden auch Gewässerdaten, wie beispielsweise Pegelstände und Wassertemperaturen, aus mehreren, heterogenen Quellen zusammengeführt.

Neben der Entwicklung des Grundsystems konzentriert sich die Arbeit auf die Qualitätsverbesserung im Open Data Service sowie allgemein in Datendiensten. Dazu wird eine Reihe von Datenqualitätsverbesserungsfiltern vorgestellt, die simple, wiederverwendbare Änderungsoperationen auf Datensätzen durchführen können. Darüber hinaus werden für den Anwendungsbereich Gewässerdaten domänenspezifische Datentypen auf Basis des „Value Object Pattern“ entworfen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ODS Open Data Service

REST Representational State Transfer

JSON JavaScript Object Notation

OSM OpenStreetMap

SI Système International d'unités

URI Uniform Resource Identifier

URL Uniform Resource Locator

API Application Programming Interface

CRUD Create, Read, Update and Delete

XML Extensible Markup Language

HTTP Hypertext Transfer Protocol

SQL Structured Query Language

WWW World Wide Web

CRC Class-Responsibility-Collaboration

UML Unified Modeling Language

SOAP Simple Object Access Protocol

1 Einleitung

Die Auswirkungen der Omnipräsenz von Heimcomputern, Notebooks und Smartphones auf unseren Alltag werden in der Gesellschaft vermehrt kritisch diskutiert. Neben den unstrittig vorhandenen Vorteilen werden als Nachteile umfassende Überwachungsmöglichkeiten und negative Folgen auf die zwischenmenschliche Kommunikation angeführt (Lin and Atkin, 2007; Webster, 2014).

Die für moderne Geräte benötigte Rechenkraft bedurfte dabei jahrzehntelanger Entwicklung. Eine bekannte Kennzahl für den technischen Fortschritt von Rechnern bietet das Mooresche Gesetz. Es besagt, dass sich die Anzahl der Transistoren und damit die potentielle Rechenleistung auf einem integrierten Schaltkreis alle zwei Jahre verdoppelt (Moore et al., 1965). In letzter Zeit hat nun vor allem die Bedeutung des Internets für unseren Alltag immer weiter zugenommen. Der Internetverkehr verdoppelt sich dabei ähnlich schnell wie die Anzahl der Transistoren. Forscher schätzen den aufgrund der Dezentralität des World Wide Webs schwerer zu berechnenden Zeitraum auf sechs Monate (IEEE 802.3 Ethernet Working Group, 2012) bis zwei Jahre (Roberts, 2000). Auch für das Gesamtdatenvolumen des Internets beträgt der Verdopplungszeitraum laut Gantz and Reinsel (2011) weniger als zwei Jahre. Dieses enorme Wachstum erhöht zum einen auf der technischen Ebene stetig die Anforderungen an effiziente Speichertechnologien und Netzwerkprotokolle. Zum anderen wird es mit der wachsenden Menge an Daten und Informationen umso wichtiger, Hilfestellungen zur effektiven Verwendung und Abfrage dieser zu geben.

Da in den letzten Jahrzehnten das Thema Klimawandel eine wichtige Rolle in der Weltpolitik spielt und auch Deutschland immer öfter von Rekordhochwassern betroffen ist (Merz et al., 2014; Te Linde et al., 2011), bietet sich das Gebiet der Gewässerforschung derzeit besonders an. Offizielle Gewässerdaten werden in Deutschland dezentral von ihren jeweiligen Bundesländern angeboten. Dies führt gerade für Menschen in Grenzregionen zu einem umständlichen Informationszugriff. Falls beispielsweise Wassersportler nicht über kritische Werte in benachbarten Bundesländern erfahren, könnten sie vermeidbaren Risiken ausgesetzt sein. Die Daten werden außerdem in vielen unterschiedlichen Formaten ausgeliefert. Oftmals bieten Portale Messwerte lediglich in Form von unstrukturierten

Tabellen an. Diese sind wiederum teilweise nur als Bilder verfügbar, was das Einlesen der Daten zusätzlich erschwert. Daher ist auch die wissenschaftliche Auswertung der Messwerte aufwendig.

1.1 Ursprüngliche Ziele der Arbeit

Diese Arbeit beschäftigt sich mit der genannten Problematik, große Mengen an Informationen sinnvoll und effizient zu nutzen. Als Beispielanwendung soll daher im Rahmen der Arbeit der Open Data Service (ODS) entstehen. Ziel dieses Dienstes ist es, vom Staat frei zur Verfügung gestellte Gewässerdaten (u.a. Pegelstände, Wassertemperaturen und Abflussvolumen) zu sammeln, aufzubereiten und dem Nutzer auf komfortable Art und Weise anzubieten. So soll eine zentrale Schnittstelle für den Zugriff auf Daten dieses Anwendungsbereichs entstehen.

Des Weiteren soll geprüft werden, wie Datenqualitätsverbesserungen innerhalb dieses Dienstes zu Vorteilen für den Nutzer führen können. Das wichtigste Datenqualitätsmerkmal ist im Rahmen dieser Arbeit die Plausibilität von Messwerten. Eine Möglichkeit, gültige Wertebereiche zu definieren, bietet das *Value Object Pattern*. Es kann beispielsweise für die Erstellung eines Datentyps *Geografische Koordinate* genutzt werden. Das *JValue*-Framework (Riehle, 2011) stellt eine Grundlage für *Value Objects* dar und wird daher für die Entwicklung dieser verwendet sowie den eigenen Bedürfnissen entsprechend angepasst. So soll eine Bibliothek von *Value Objects* für mögliche Messwert-Datentypen im Bereich Gewässerdaten implementiert werden.

1.2 Anpassungen der Ziele der Arbeit

Eine durchgängig konkrete Implementierung von *Value Objects* für alle vorhandenen und zu erwartenden Messwerte ist im Rahmen des Open Data Service bislang nicht sinnvoll, da nur ein Funktionsbaustein des *Value Object Pattern*, die Definierbarkeit von Wertebereichen, genutzt wird. In der Arbeit werden die *Value Objects* stattdessen hauptsächlich dynamisch definiert und auf diese Weise die Plausibilität der Daten geprüft. Des Weiteren konzentriert sich die Arbeit nun auch auf die Erarbeitung von individuell konfigurierbaren Datenqualitätsfiltern, die auf beliebige Datensätze angewendet werden können. Diese sollen einfache Operationen beispielsweise zur Umstrukturierung von Datenstrukturen durchführen.

2 Forschung

Das folgende Kapitel stellt den Hauptteil dieser Arbeit dar. In Abschnitt 2.1 werden verwandte Arbeiten im Bereich Datenqualität vorgestellt. 2.2 befasst sich mit den Anforderungen und Problemstellungen, die es zu erfüllen gilt. In Abschnitt 2.3 wird der Lösungsansatz für die Forschungsfrage beschrieben. 2.4 präsentiert als Ergebnisse der Arbeit die Implementierungsdetails anhand eines konkreten Software-Entwurfs. Als nächstes werden diese in 2.5 kritisch diskutiert und ein Ausblick auf mögliche Weiterentwicklungen gegeben. In 2.6 wird die Arbeit schließlich zusammengefasst.

2.1 Verwandte Arbeiten

In diesem Abschnitt werden die Grundlagen der Arbeit vorgestellt. Zunächst werden in 2.1.1 theoretische Hintergründe zum Thema Datenqualität beschrieben. Anschließend befasst sich 2.1.2 mit REST sowie Diensten, die diesen Architekturstil einsetzen.

2.1.1 Datenqualität

2.1.1.1 Definition

In dieser Arbeit ist sowohl von *Daten* als auch von *Informationen* die Rede. Unter *Daten* versteht man eine Sammlung von Symbolen. Von *Informationen* spricht man dagegen, wenn *Daten* interpretiert werden und sie dadurch eine Bedeutung erhalten. Betrachtet man beispielsweise eine beliebige niedergeschriebene Zahl, so kann es sich dabei um eine Telefonnummer, eine Kontonummer, ein Passwort oder auch eine Zufallszahl handeln. Erst das Wissen um die korrekte Interpretation der Zahl macht die *Daten* zu *Informationen*. Umgekehrt kann man *Informationen* in Form von *Daten* kodieren (Aamodt and Nygård, 1995; Bellinger et al., 2004). Dieser Zusammenhang ist in Abbildung 2.1 dargestellt.

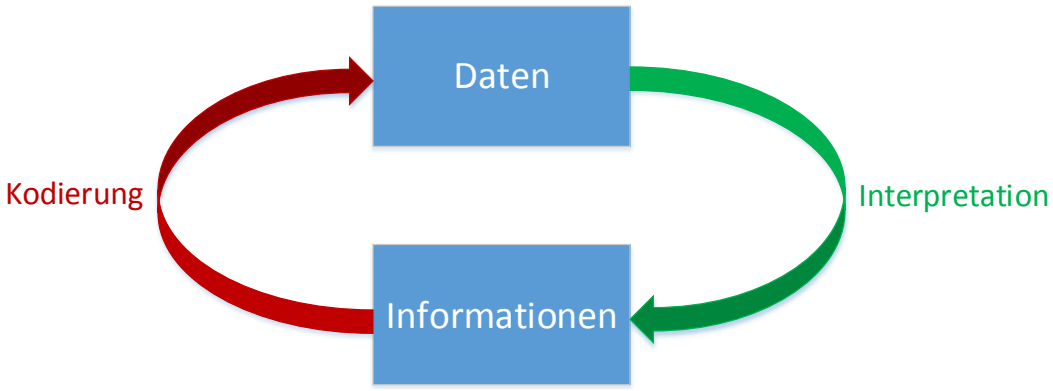


Bild 2.1: Zusammenhang zwischen Daten und Informationen

Für den Begriff der *Qualität* existieren des Weiteren viele verschiedene Definitionen. Die Qualitätsmanagement-Norm ISO 9000 (2005) definiert *Qualität* als den Grad der Erfüllung von Anforderungen durch spezifische Merkmale.

Mit *Datenqualität* bezeichnet man schließlich ein Gütemaß für Daten, das in eine Reihe von Einzelkriterien aufgeteilt ist. In Anlehnung an die Definition der *Qualität* ergibt sich, dass die Datenqualität wiederum direkt proportional zum Erfüllungsgrad der Kriterien ist. Nach Wang and Strong (1996) werden die Datenqualitätskritierien in vier Kategorien eingeteilt.

1. Intrinsische Datenqualität:

   Glaubwürdigkeit

   Genauigkeit

   Objektivität

   Reputation (Vertrauenswürdigkeit der Quelle)

2. Kontextbezogene Datenqualität:

   Mehrwert

   Relevanz (Nutzbarkeit der Daten)

   Aktualität

   Vollständigkeit

   Datenmenge (sinnvolle Einschränkung der Datenmenge)

3. Repräsentationsbezogene Datenqualität:

   Interpretierbarkeit (Korrektheit der Sprache und Einheiten)

   Verständlichkeit (Widerspruchsfreie, verständliche Daten)

   Widerspruchsfreie Darstellung (Einheitliche Darstellung von gleichen Datentypen)

   Präzise Darstellung (Kompakte Darstellung ohne Verzicht auf essentielle Informationen)

4. Zugriffsbezogene Datenqualität:

   Zugänglichkeit (Verfügbarkeit der Daten zu jedem Zeitpunkt)

   Zugriffssicherheit (Möglichkeit der Authentifizierung und Autorisierung beim Datenzugriff)

Zu Beginn dieses Abschnitts wurde der Unterschied zwischen den Begriffen der *Daten* und der *Informationen* dargestellt. In der weiteren Arbeit wird jedoch, wie häufig in der Literatur (vergleiche die Publikationen des selben Autors Wang and Strong (1996) und Wang (1998)), nicht zwischen Informations- und Datenqualität unterschieden.

2.1.1.2 Bekannte Ansätze zur Verbesserung der Datenqualität

Das *JValue*-Framework

Eine Möglichkeit, Datenqualität in Softwaresystemen zu erhöhen, bietet das *Value Object Pattern*. Das Ziel des Musters ist es, Objekte zu erschaffen, die dem Verhalten von primitiven Datentypen wie *Integer* oder *Float* gleichen. *Value Objects*, deren Typ *Value Type* genannt wird, sind nicht veränderbar. Es dürfen dem Client also keine den Zustand beeinflussenden Methoden angeboten werden. Im Gegensatz zu normalen Klassen wird bei *Value Types* die Gleichheit zweier Objekte nicht durch eine ID bestimmt, sondern lediglich durch die Werte der Felder. Sind alle Werte gleich, so sind dies auch die jeweiligen *Value Objects* (Cunningham, 1995; Fowler, 2002; Riehle, 2006).

Ein Beispiel für einen *Value Type* liefert die Umsetzung einer „Geld-Klasse“ (Riehle, 2006). Neben Kontostand und Währungskennzeichen enthält diese auch Methoden zum Einzahlen sowie zum Abheben von Geldbeträgen. Ein konkreter Vorteil hinsichtlich der Datenqualität ist hierbei die Möglichkeit, bei Auszahlungen auf einen ausreichenden Kontostand zu prüfen. Ähnlich dazu können beispielsweise Messwerte implementiert und so deren Plausibilität festgestellt werden.

Microsofts objektorientierte Programmiersprache C# setzt das Muster bereits für alle definierten *Structs* ein (ISO/IEC 23270, 2006). Auch in Java ist eine fest integrierte Unterstützung von *Value Objects* in Zukunft geplant[1].

Eine Möglichkeit, *Value Objects* bereits jetzt in Java zu verwenden, bietet *JValue*. Im Rahmen dieses Projekts wurde ein grundlegendes Wertetypmodell umgesetzt. Es existieren Klassen für einfache Datentypen wie *String* und *Number*, aber auch für Container wie *List*. Ebenso wurden Typen für Datumsangaben, hierarchische Namen und SI-Einheiten (Thompson and Taylor, 2008) implementiert. Zugelassene Werte der Felder können durch verschiedene „Einschränkungsklassen“, deren Instanzen einen konkreten Wert oder einen Wertebereich beschreiben, definiert werden (Bäumer et al., 1998). Die aktuelle Fassung des Projekts bildet die Grundlage für die im Laufe dieser Arbeit entwickelten *Value Types*.

2.1.2 REST

2.1.2.1 Definition

Representational State Transfer (REST) ist eine Architekturform für Web-Anwendungen, die von einem der Entwickler des „Hypertext Transfer Protocol (HTTP)“-Protokolls, Roy Fielding, im Rahmen seiner Dissertation (Fielding, 2000) spezifiziert wurde. In dieser leitete Fielding Grundprinzipien von HTTP ab, um die Idee für zukünftige Dienste wiederverwendbar zu machen. Das auf HTTP basierende Internet ist daher das bekannteste Beispiel einer REST-Schnittstelle. Um den Prinzipien von REST zu entsprechen, muss ein Interface folgende vier Eigenschaften aufweisen:

Identifizierbarkeit der Ressourcen

Alle Ressourcen müssen eindeutig identifizierbar sein. Im HTTP-Protokoll wird dies durch Zuweisung einer URI erreicht.

Interaktion mit Ressourcen durch einheitliche Schnittstelle

Es existiert eine simple, einheitliche Schnittstelle, die eine Menge von Operationen auf Ressourcen durchführen kann. Im Fall von HTTP umfasst dies unter anderem HTTP GET,

1 `http://openjdk.java.net/jeps/169`

HTTP POST, HTTP PUT, HTTP DELETE und HTTP HEAD (Fielding and Reschke, 2014).

Selbstbeschreibende Nachrichten

Die Nachrichten müssen alle Informationen enthalten, die zur Interpretation dieser nötig sind.

Möglichkeit der Verlinkung von Ressourcen

Es muss die Möglichkeit bestehen, zwischen verschiedenen Ressourcen navigieren zu können, beispielsweise via Hyperlinks.

Sind die eben beschriebenen vier Eigenschaften erfüllt, so ist eine zustandslose Interaktion zwischen Client und Server möglich. Die Schnittstelle wird dann als *RESTful* bezeichnet (Fielding and Taylor, 2002; Pautasso et al., 2008; Richardson and Ruby, 2008).

2.1.2.2 Beispiele für RESTful Web Services

In diesem Teilabschnitt werden einige RESTful Web Services vorgestellt. Dabei dienen sowohl OpenStreetMap (OSM) als auch *PEGELONLINE* später als Datenquellen für den entwickelten Open Data Service.

OpenStreetMap

OSM[1] ist ein freies Geoinformationssystem, das mithilfe freiwilliger Projektteilnehmer geografische Daten der gesamten Welt sammelt (Haklay, 2010; Haklay and Weber, 2008). Es steht damit hauptsächlich in Konkurrenz zum lizenztechnisch strenger regulierten *Google Maps*[2] (O'Reilly, 2007).
OSM basiert auf drei Typen von Ressourcen:

Nodes Punkte auf der Karte, beispielsweise Häuser

Ways Eine Sammlung zusammengehöriger *Nodes*, z.B. eine Straße

Relations Eine Sammlung zusammengehöriger *Nodes* und *Ways*, beispielsweise das deutsche Autobahnnetz

1 `http://www.openstreetmap.org`
2 `https://www.google.de/maps`

Möchte man zu OpenStreetMap beitragen, so geschieht dies über deren REST-Schnittstelle[1]. Sie umfasst die grundlegenden „Create, Read, Update and Delete (CRUD)“-Datenbankoperationen. Die Payloads der Anfragen und Antworten enthalten Daten im „OSM Extensible Markup Language (XML)“-Format, das auf einem für OSM spezifizierten *XML Schema* aufbaut.

Für den wiederholten rein lesenden Zugriff werden jedoch in der Regel komprimierte Gesamtdatensätze der betreffenden Gebiete verwendet. Es existieren beispielsweise je nach gewünschtem Umfang Datensammlungen für Mittelfranken, Bayern, Deutschland, Europa und für die ganze Welt. Um die Datenmenge zu reduzieren, werden für viele der Datensätze auch *Changesets* angeboten, welche die Änderungen zur Vorversion im *OsmChange*-Format[2] enthalten. Die Nutzung der Download-Pakete ist für beide Seiten vorteilhaft. Einerseits verringert sich die Rechen- und Netzwerklast auf den OSM-Servern, da die Zahl der Anfragen sinkt und die Datensammlungen einfacher verteilt werden können. Andererseits wird auch der Client während seiner eigenen Berechnungen unabhängig von der Kommunikation mit dem Server.

PEGELONLINE

Eine weitere Quelle des Open Data Service, die gleichzeitig dem vorgestellten REST-Paradigma entspricht, ist das *Gewässerkundliche Informationssystem der Wasser- und Schifffahrtsverwaltung des Bundes, PEGELONLINE*[3]. Dieser Dienst bietet Gewässerdaten einer Vielzahl von Messstationen in Deutschland an. Dies umfasst beispielsweise Informationen zu Pegelständen, Wassertemperaturen und elektrischen Leitwerten. Neben der Möglichkeit des REST-Zugriffs können *PEGELONLINE*-Daten unter anderem auch durch Verwendung von SOAP abgefragt werden. Die gemessenen Werte der Stationen bleiben 30 Tage lang online verfügbar. Innerhalb dieser Zeit kann die Entwicklung der Messwerte auch direkt durch einen REST-Aufruf visualisiert werden.

Soziale Netzwerke

Ein weiteres Beispiel für einen bekannten, auf REST basierenden, Web-Dienst ist die Mikroblogging-Plattform *Twitter*[4]. Die verwendeten Ressourcen umfassen unter anderem

1 `http://wiki.openstreetmap.org/wiki/API_v0.6`
2 `http://wiki.openstreetmap.org/wiki/OsmChange`
3 `https://www.pegelonline.wsv.de`
4 `https://www.twitter.com`

Tweets, *Timelines* (Sammlungen von *Tweets*), Nutzer, Freundeslisten und Suchergebnisse. Die einfache Schnittstelle basiert lediglich auf HTTP GET und HTTP POST, wobei beispielsweise die von HTTP DELETE gewohnte Löschfunktionalität ebenfalls über „HTTP POST"-Anfragen realisiert wird (Go et al., 2009; Makice, 2009).

Auch *Facebook*[1] setzte lange Zeit auf eine REST-Schnittstelle. Diese wurde zwischenzeitlich jedoch von der *Graph API* abgelöst (Ko et al., 2010).

2.1.3 Dokumentenorientierte Datenbanken

Dokumentenorientierte Datenbanken sind eine Form von *NoSQL*-Datenbanken. Unter *NoSQL* versteht man eine Sammlung von Paradigmen, die Alternativen zur klassischen relationalen Speicherung von Daten darstellen. Die bekanntesten davon sind, neben dokumentenorientierten Datenbanken, objektorientierte Datenbanken (Saake et al., 1997), Graphdatenbanken (Angles and Gutierrez, 2008) und Key-Value-Stores (z.B. Dynamo: (DeCandia et al., 2007)).

Im Gegensatz zu relationalen Datenbanken verwenden dokumentenorientierte Datenbanken als Grundeinheiten keine Tabellen, sondern Dokumente. Ein Dokument entspricht auf der obersten Ebene einer, aus Programmiersprachen bekannten, Map und besteht daher aus einer Menge von Schlüssel-Wert-Paaren, wobei es sich beim Schlüssel immer um eine Zeichenkette handeln muss. Die Werte können hingegen sowohl primitive Datentypen als auch Container-Objekte darstellen.

Die Vorteile bestehen darin, dass Dokumente zum einen schemafrei sind und dadurch eine flexible Anzahl und Zusammenstellung von Feldern enthalten können. Zum anderen sind sie im Gegensatz zu Tabellen hierarchisch aufgebaut. Deswegen können zusammengehörige Daten in der Regel durch ein einziges Dokument repräsentiert werden. Für Softwaresysteme bedeutet dies, dass auch deren Objektinstanzen mithilfe eines einzelnen Dokuments abgespeichert werden (Cattell, 2011; Stonebraker, 2010). Einige der bekanntesten Beispiele für dokumentenorientierte Datenbanken sind *MongoDB*[2], *CouchDB*[3] und *IBM Notes*[4].

1 `https://www.facebook.com`
2 `https://www.mongodb.org/`
3 `http://couchdb.apache.org/`
4 `http://www-03.ibm.com/software/products/de/ibmnotes`

2.2 Forschungsfrage

In diesem Abschnitt der Arbeit wird die Anforderungsanalyse durchgeführt. Zunächst werden in 2.2.1 die Anforderungen an das Grundsystem, anschließend in 2.2.2 die Ziele für die Datenqualitätsverbesserung beschrieben. Die nicht-funktionalen Anforderungen an den ODS befinden sich in Ausarbeitungsabschnitt 3.1.1.

2.2.1 Funktionale Anforderungen an den Open Data Service

Benutzerfreundliche REST-Schnittstelle

Die Architektur des Open Data Service soll den in 2.1.2 beschriebenen Prinzipien von REST entsprechen. Der Client sollte Ressourcen durch ein einfaches HTTP GET, welches auch ein Browser bei Seitenaufrufen ausführt, abfragen können. Wenn die gleiche Adresse zu einem späteren Zeitpunkt erneut aufgerufen wird, muss es sich um die gleichen Daten, möglicherweise in einer aktualisierten Form, handeln. Die abfragbaren Daten sollen des Weiteren in einem einheitlichen Format angeboten werden.

Falls es vorliegt, soll außerdem das Schema der angebotenen Daten abfragbar sein. In diesem werden Struktur und Datentypen des Dokuments beschrieben. Ebenso soll die Möglichkeit bestehen, sich mit einem Aufruf alle möglichen Abfragen und damit die gesamte Schnittstelle auflisten zu lassen.

Konzepte zur Speicherung und Verwaltung der Daten

Neben klassischen SQL-Datenbanken werden in der heutigen Zeit immer häufiger *NoSQL*-Datenbanken (vergleiche 2.1.3) eingesetzt. Es gilt zu evaluieren, welche Datenbanktechnologie zur Speicherung der Daten sinnvoll ist. Die Nutzeranfragen müssen dabei möglichst schnell verarbeitet werden können. Außerdem soll der Programmieraufwand so gering wie möglich gehalten werden.

Identifikation geeigneter Datenquellen

Der ODS stellt einen Dienst zum Zusammenführen und komfortablen Anbieten heterogener Datenquellen dar. Im Rahmen dieser Arbeit werden Daten aus dem Bereich Wasser und Geographie verwendet. Hierfür finden sich im Internet viele verschiedene Quellen, welche sich jedoch hinsichtlich ihrer Schnittstellen und Benutzbarkeit stark unterscheiden. Jedes Bundesland bietet beispielsweise eigene Daten zur Hochwasserwarnung an. Die In-

formationen werden je nach Ursprung auch in vielen verschiedenen Formaten dargestellt. Hier gilt es herauszufinden, welche Datensätze komfortabel abzurufen sind, wodurch für zukünftige Anwendungen wiederverwendbare Zugriffsmethoden entwickelt werden können.

Regelmäßige Informations-Updates

Der Open Data Service soll in regelmäßigem Abstand prüfen, ob die verwendeten Quellen neue Daten anbieten, und diese dann wenn nötig in den ODS aufnehmen. Die Update-Intervalle sollen je nach Quellumfang individuell konfigurierbar sein.

2.2.2 Anforderungen an die Datenqualitätsverbesserung

Umsetzung einer Sammlung von *Value Object Types* im Bereich Gewässerdaten

Da der ODS im Rahmen dieser Arbeit mit Gewässerdaten bestückt wird, soll im Einklang dazu eine Reihe von *Value Object Types* für diesen Anwendungsbereich erstellt werden. Es gilt festzustellen, welche Datentypen in vielen verschiedenen Datensätzen vorkommen und sich daher für eine Abstraktion anbieten.

Ein Beispiel für einen *Value Object Type* im Bereich Gewässerdaten sind Koordinaten, welche aus geografischer Breite und geografischer Länge zusammengesetzt sind. Die Breite kann dabei Werte von -90° (am Südpol) über 0° (Äquator) bis +90° (Nordpol) annehmen. Für die geografische Länge wurde ein künstlicher Nullmeridian in Greenwich, London festgelegt. Es sind davon ausgehend Werte von -180° (in westlicher Richtung) bis +180° (östliche Richtung) möglich. Diese Wertebereiche sollen durch einen *Value Object Type* validiert werden.

Entwicklung von Datenqualitätsfiltern

Zur Steigerung der Datenqualität im Open Data Service soll eine Reihe von speziellen Filtern entwickelt werden, die jeweils kleine, universelle Änderungsoperationen durchführen. Diese Filter werden dabei so entworfen, dass sie auf verschiedene Datensätze anwendbar sind. Ein Durchlauf der Filter soll den Erfüllungsgrad der in 2.1.1 beschriebenen Datenqualitätskriterien dauerhaft erhöhen können.

2.3 Forschungsansatz

Dieser Teil der Arbeit behandelt den Lösungsansatz für die umzusetzenden Anforderungen. Zunächst wird in 2.3.1 die geplante Architektur des Open Data Service vorgestellt. Anschließend folgt in Abschnitt 2.3.2 die Planung der Datenqualitätsverbesserungen.

2.3.1 Konzept des Open Data Service

In diesem Abschnitt wird die Grundidee für die Architektur des Open Data Service beschrieben. Der ODS ist im Wesentlichen in zwei Hauptkomponenten aufgeteilt. Dabei handelt es sich zum einen um die Serverkomponente, die in 2.3.1.1 vorgestellt wird. Der zweite Teil ist die Importkomponente, deren Struktur Abschnitt 2.3.1.2 behandelt. Abschließend wird in 2.3.1.3 ein Gesamtüberblick über die Architektur des Open Data Service gegeben.

2.3.1.1 Serverkomponente

Die Serverkomponente dient der Bereitstellung der Daten des Open Data Service gegenüber Clients. Abbildung 2.2 beschreibt die Grobstruktur dieser Komponente anhand einer beispielhaften Befehlskette. Der Benutzer kann verschiedene Anfragen an die REST-Schnittstelle des Servers (rot gekennzeichnet) stellen. Zum einen existieren Anfragen, die der Server ohne Zugriff auf die Datenbank bearbeiten kann, beispielsweise das Anfordern einer Übersicht über alle möglichen Befehle. Die Abbildung beschreibt jedoch den häufigsten Anwendungsfall: die Abfrage von Datensätzen des Open Data Service. Fordert der Client Daten an, so erfolgt die Datenbeschaffung innerhalb des ODS über eine abstrahierte Datenzugriffsschicht (lila im Bild), die das System von einer konkreten Datenbankimplementierung entkoppelt. Möchte man später auf eine relationale Datenbank umsteigen, muss so nur die unterste Schicht angepasst werden. Die Zugriffsschicht kennt die darunter liegende Datenbank und führt spezifische Abfragen auf dieser aus, die aus Effizienzgründen vordefiniert sein können.

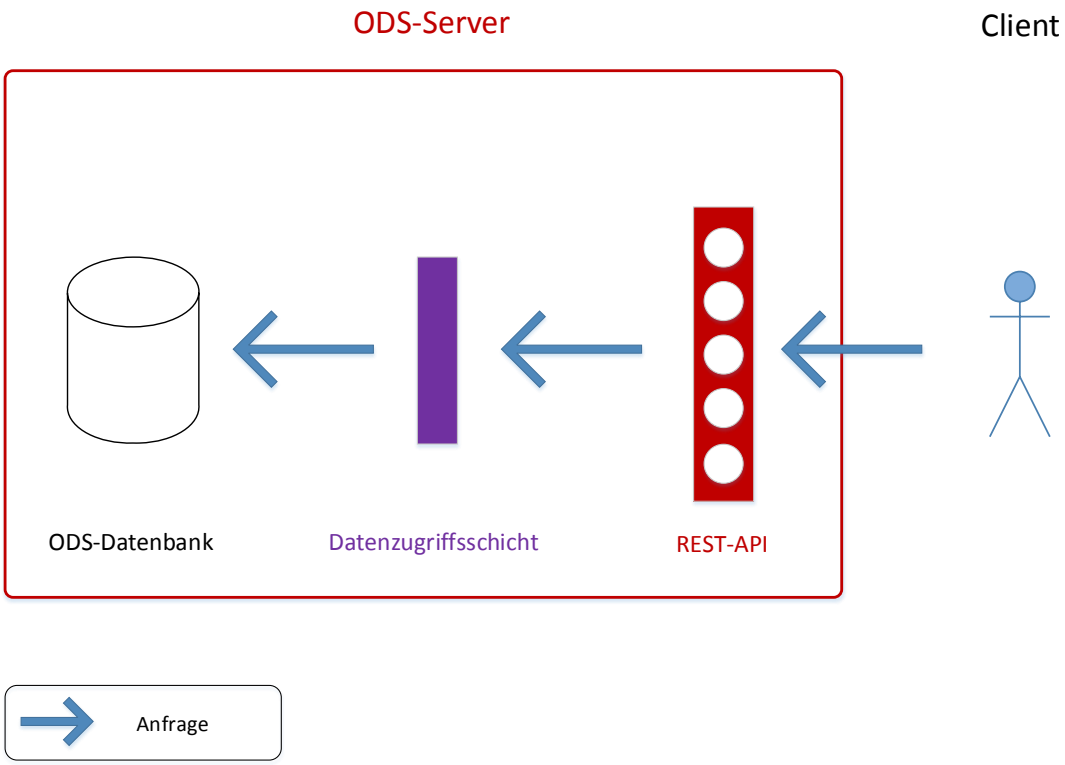


Bild 2.2: Anfrageverarbeitung der Serverkomponente

Der ODS-Server kann, je nach Anwendungszweck, im lokalen Netzwerk oder im Internet betrieben werden. Als Testbeispiel wird eine Instanz des ODS auf einem öffentlichen Tomcat-Server[1] der Universität Erlangen-Nürnberg installiert.

Die Beispielimplementierung des Open Data Service benutzt als Datenbank *CouchDB*. Es wurde sich in dieser Arbeit für eine dokumentenorientierte Datenbank entschieden, da die Datensätze, die im Bereich Gewässerdaten vorhanden sind, Dokumenten ähneln und so intuitiv ohne große Anpassungen verwendet werden können. *CouchDB* (Apache License, Version 2.0) hat im Vergleich zu *MongoDB* (GNU AGPL v3.0) eine weniger restriktive Lizenz, weshalb die Wahl im Rahmen des Projekts auf diese fiel. *CouchDB* nutzt zur Speicherung von Objekten das „JavaScript Object Notation (JSON)“-Format, welches auch für die Auslieferung der Daten an die Benutzer des ODS eingesetzt wird.

Der interne Zugriff auf die *CouchDB*-Datenbank erfolgt nativ ebenfalls über eine REST-Schnittstelle. Zur komfortableren Programmierung wird jedoch im Open Data

1 `http://tomcat.apache.org/`

Service die Java-Bibliothek *Ektorp*[1] verwendet. Diese abstrahiert die REST-Schnittstelle vom Code des ODS und bietet so eine einfache Schnittstelle an, welche die CRUD-Operationen umsetzt. Die prototypische Implementierung der ODS-Datenzugriffsschicht basiert auf den Funktionen dieser Bibliothek.

Zur Implementierung der REST-Schnittstelle des Open Data Service wird ebenfalls eine Hilfsbibliothek, *Restlet*[2], benutzt.

2.3.1.2 Importkomponente

Der zweite Hauptbestandteil des Open Data Service ist die Importkomponente. Deren Aufgabe ist es, die Datenbank des ODS mithilfe externer, freier Datenquellen zu füllen. Außerdem wird regelmäßig geprüft, ob die bekannten Quellen aktualisierte Daten zur Verfügung stellen, um diese gegebenenfalls in die Datenbank einpflegen zu können.

Abbildung 2.3 beschreibt den Grundaufbau der Importkomponente anhand eines einfachen Datenflussdiagramms. Der Import besteht aus einer Reihe von Bearbeitungsschritten. Um auf die externen Daten zugreifen zu können, wird eine quellspezifische Adapterschicht (Schritt 1, rot gekennzeichnet) entwickelt. Diese umfasst zum einen verschiedene, formatabhängige Zugriffsbausteine wie z.B. einen JSON- und einen XML-Grabber. Falls die Daten nicht im innerhalb des ODS verwendeten JSON-Format vorliegen, werden diese hier außerdem in dieses konvertiert. Diese JSON-Daten durchlaufen nun eine Filterkette, in der sie ergänzt (Schritt 2, blau), aufbereitet (Schritt 4, grün) und abgespeichert (Schritte 3 und 5 für Roh- respektive verbesserte Daten, orange) werden. Der Zugriff auf die Datenbank erfolgt, wie beim Server, über die lila gekennzeichnete Datenzugriffsschicht. Die genaue Struktur der Filterkette wird im folgenden Teilabschnitt beschrieben.

1 `http://ektorp.org/`
2 `http://restlet.com/`

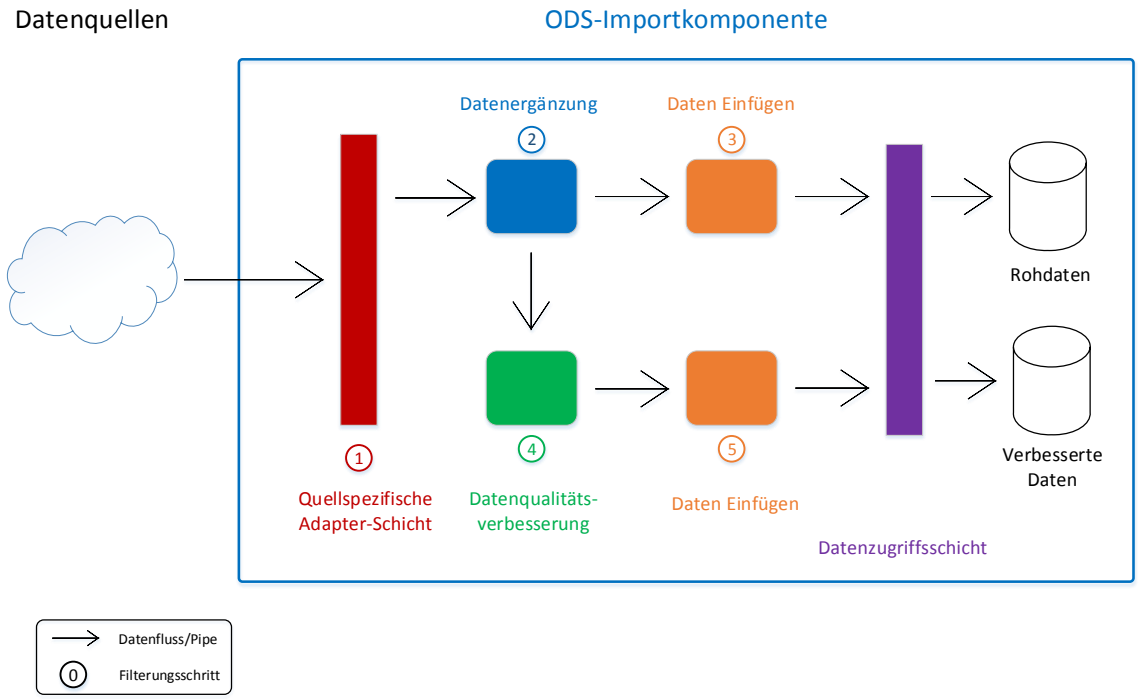


Bild 2.3: Datenfluss der Importkomponente

Filterkette

Der Import von Datenquellen soll im ODS so umgesetzt werden, dass jede einzelne Komponente einen Filterschritt darstellt. Dazu wird eine Abwandlung des „*Pipes and Filters*“-Musters (Buschmann et al., 2007; Garlan and Shaw, 1994) verwendet.

Nicht jeder Schritt in der Kette stellt dabei im klassischen Sinn eine Filterung dar. Zum Beispiel handelt es sich beim Herunterladen von freien Datensätzen um keinen konkreten Filterungsschritt. Aus Abstraktionsgründen werden hier jedoch alle bearbeitenden Komponenten unter dem Begriff Filter zusammengefasst. In der zu entwickelnden Software wird der Zusammenhang durch die Implementierung einer gemeinsamen Schnittstelle deutlich.

Abbildung 2.4 beschreibt einen typischen Einsatz der Filterkette für den Import neuer oder aktualisierter Datensätze. Die Datensätze werden dabei Schritt für Schritt durch Pipes weitergereicht. Die letzten beiden Filterungen werden nur bei Quelldaten, für die Maßnahmen zur Qualitätsverbesserung konfiguriert sind, durchgeführt.

- Der erste Bearbeitungsschritt (rot gekennzeichnet) ist das Einlesen der Quelldaten und die Umwandlung in ein JSON-kompatibles Format.

- Als zweites (blau) wird ein Filter eingesetzt, der die Datensätze um zusätzliche Attribute ergänzt. Dies ist beispielsweise nötig, um den ODS-internen Typ des Datensatzes zur späteren Identifikation festzulegen.
- Schritt drei (orange) fügt die Rohdaten in die Datenbank ein.
- In Schritt vier (grün) können die Daten mithilfe einer beliebig kombinierbaren Menge von Datenqualitätsfiltern aufbereitet werden.
- Diese verbesserten Daten müssen im abschließenden, fünften Schritt (orange) wiederum in die Datenbank eingefügt werden. Es wird der gleiche Filter wie in Schritt drei verwendet.

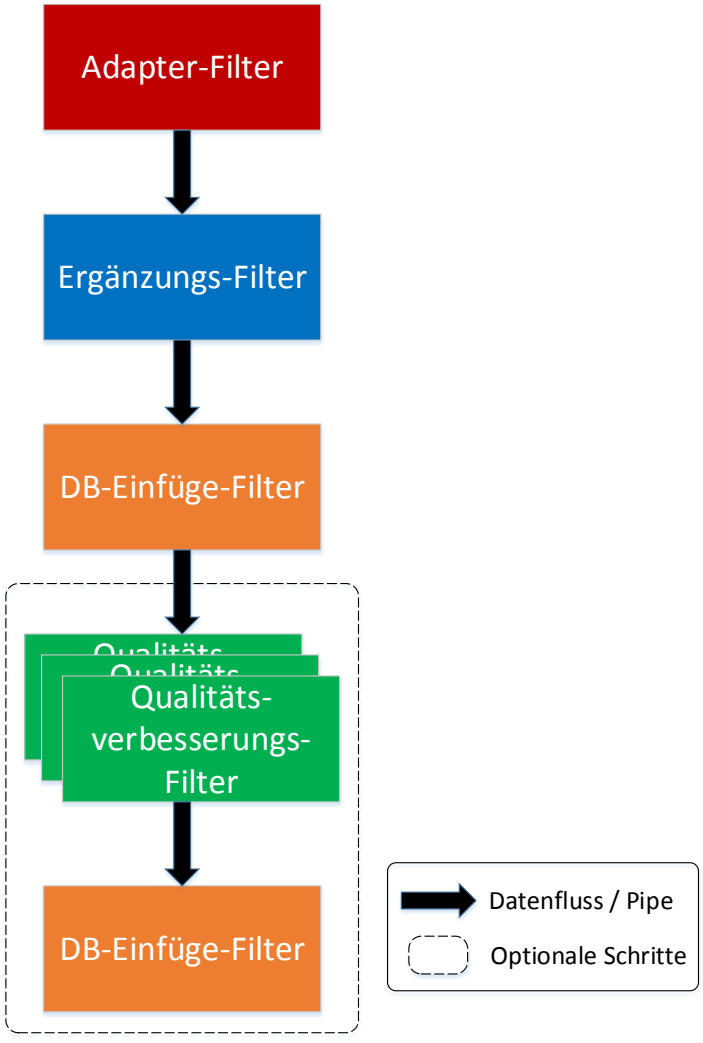


Bild 2.4: Filterkette für den Datenimport

2.3.1.3 Gesamtarchitektur

In den vorigen Abschnitten wurde die strukturelle Trennung zwischen Server- und Importkomponente beschrieben. Das Diagramm 2.5 fasst diese nun zusammen und gibt so einen Gesamtüberblick über das geplante System. Die Pfeile sind in Richtung des Datenflusses gezeichnet. In der Skizze ist die Importkomponente blau, die Serverkomponente rot umrandet. Auch die sonstigen Architekturbestandteile sind in den bereits zuvor verwendeten Farben dargestellt. Wie aus dem Diagramm ersichtlich, kann der Client sowohl auf Rohdaten als auch auf verbesserte Daten des Servers zugreifen. Datenbank und Datenzugriffsschicht sind Teil beider Komponenten, da sie sowohl beim Einfügen von Datensätzen als auch beim späteren Auslesen dieser benötigt werden. Die Zugriffsschicht ist lediglich aus Anschaulichkeitsgründen doppelt abgebildet. Sie existiert im System nur einmal.

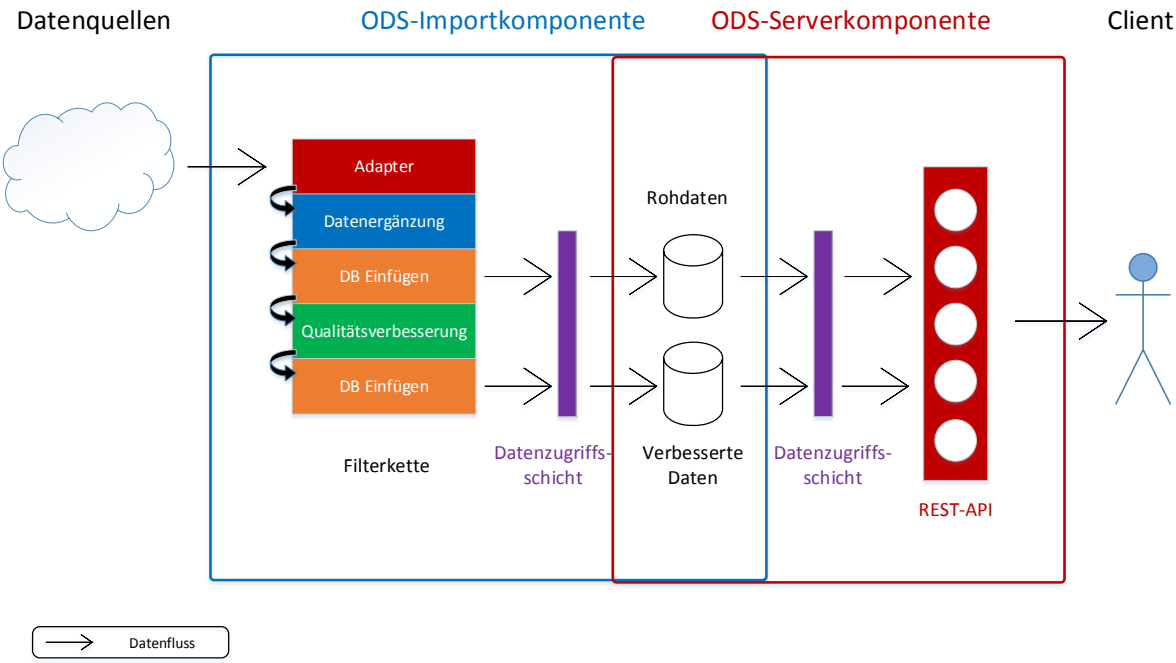


Bild 2.5: Gesamtarchitektur des Open Data Service

2.3.2 Konzept der Datenqualitätsverbesserung

In diesem Abschnitt wird das Konzept der Qualitätsverbesserungsmaßnahmen präsentiert. 2.3.2.1 zeigt den Grobentwurf der generischen Datenqualitätsfilter. In 2.3.2.2 folgt schließlich die Erarbeitung der umzusetzenden *Value Objects*.

2.3.2.1 Qualitätsverbesserungsfilter

Kombinationsfilter

Viele Datenquellen im Internet haben eine flache Hierarchie. Das bedeutet, dass die Attribute in einem Quelldokument ungeordnet und unzusammenhängend aufgelistet sind. Beispielsweise existieren bei dem Dienst *PEGELONLINE* Datensätze für viele verschiedene Pegelmessstationen in Deutschland (vergleiche 2.1.2.2). Jeder dieser Datensätze besitzt jeweils ein Attribut für die geografische Breite und eines für die geografische Länge des Stationsstandorts. Dass die beiden Attribute zusammen die Koordinaten der Messstation darstellen, ist nicht spezifiziert. Diese Information bringt den Nutzern des Open Data Service jedoch einen Mehrwert.

Dies soll daher mit einem ersten Filter ermöglicht werden, welcher eine beliebige Menge von Attributen unter einem abstrahierenden Namen zusammenfasst. Zur Veranschaulichung wurde das Diagramm 2.6 erstellt. Es vergleicht in der Form eines Klassendiagramms die Struktur eines ursprünglichen und eines verbesserten Stations-Dokuments. Die rot gekennzeichneten Felder für den Längengrad und Breitengrad der Station werden in einen neuen Typ *Koordinate* ausgelagert, dessen Instanz (grün) in einem aufbereiteten Stations-Dokument an deren Stelle tritt. Es ist zu beachten, dass es sich hier trotz der Verwendung eines Klassendiagramms nicht um konkret existierende Klassen handeln muss. Da sich der Aufbau von Klassen und JSON-Dokumenten ähnelt, wurde diese Darstellungsform verwendet.

Obwohl die beschriebenen Datenstrukturen zunächst nur JSON-Objekte repräsentieren, können die entstandenen Extrakte jedoch später als *Value Objects* umgesetzt werden. Die verschiedenen Datenqualitätsmaßnahmen sind in diesem Punkt also miteinander verknüpft.

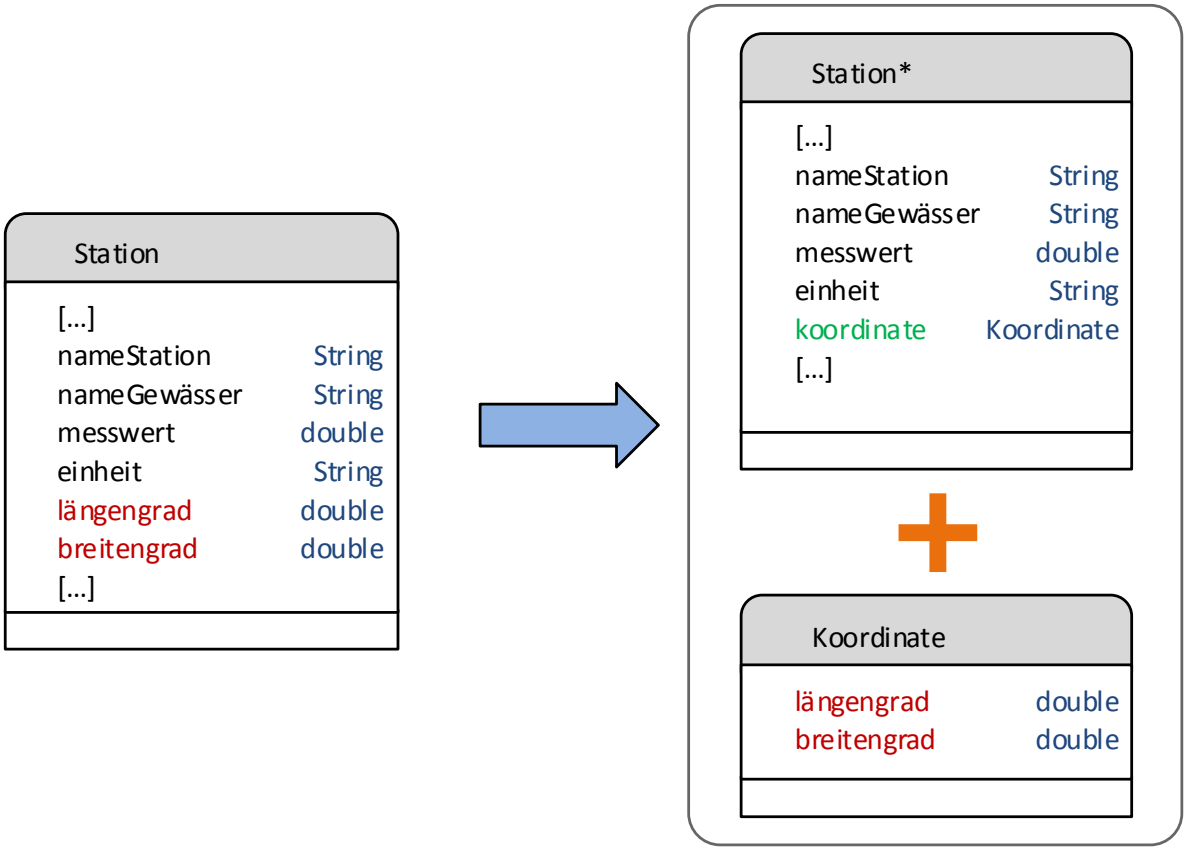


Bild 2.6: Skizze des Kombinationsfilters

Umbenennungsfilter

Bei der Vielzahl von Quellen im Internet kommt es häufig vor, dass gleichen Datentypen unterschiedliche Bezeichnungen gegeben werden. In der Beispielanwendung des Open Data Service werden Pegelstände sowohl vom Service *PEGELONLINE* als auch vom *Pegelportal Mecklenburg-Vorpommern*[1] bezogen. Der genaue Name der Messstation befindet sich einmal im Feld *longname* und das andere Mal im Feld *station*. Analog dazu wird die Einheit des gemessenen Pegelstandes einmal mit *unit* und das andere Mal mit *levelUnit* bezeichnet. Selbst Unterschiede in der Groß- und Kleinschreibung oder in der Worttrennung können jedoch Probleme verursachen, falls diese nicht erkannt und gegebenenfalls korrigiert werden. Daher wird ein simpler Umbenennungsfilter entwickelt, der das Ziel hat, Feldernamen in Dokumenten zu vereinheitlichen.

1 `http://www.pegelportal-mv.de/pegel_mv.html`

2.3.2.2 Umsetzung der *Value Objects*

Wie eingangs bereits erwähnt, werden die *Value Objects* nicht als vollständige, konkret implementierte Klassenbibliothek umgesetzt. Dies geschieht nur für sehr häufig vorkommende Datentypen wie *Koordinate*, welche als Abstraktion innerhalb des ganzen ODS verwendet werden sollen. Der Grund ist, dass im Rahmen des ODS bislang lediglich die Prüfung des Wertebereichs relevant ist und die anderen Vorteile des in 2.1.1.2 vorgestellten *Value Object Pattern* zum jetzigen Zeitpunkt nicht benötigt werden. Daher ist eine durchgängige Implementierung konkreter Klassen nicht sinnvoll. Stattdessen werden die Value Objects hauptsächlich dynamisch umgesetzt. Das bedeutet, dass die gültigen Werte wichtiger Datentypen zur Laufzeit dynamisch definiert und direkt für die Validierung der Dokumente verwendet werden.

2.4 Implementierung

In diesem Abschnitt wird die prototypische Umsetzung des Open Data Service vorgestellt. Zunächst erfolgt in 2.4.1 eine Beschreibung des konkreten Software-Entwurfs des Grundsystems. 2.4.2 behandelt anschließend die Implementierungsdetails der Datenqualitätsverbesserungskomponenten.

2.4.1 Implementierung des Open Data Service

Die folgenden Teilabschnitte beschreiben die Implementierung des Open Data Service. Diese wurde im Rahmen mehrerer Abschlussarbeiten an der Forschungsgruppe für Open-Source-Software der Universität Erlangen-Nürnberg durchgeführt. Neben „Unified Modeling Language (UML)“-Diagrammen werden die Klassen zusätzlich mithilfe von „Class-Responsibility-Collaboration (CRC)“-Karten (Beck and Cunningham, 1989) vorgestellt.

Abbildung 2.7 zeigt den Prototyp einer CRC-Karte. Diese sind aus drei Bereichen aufgebaut, wobei die Kopfzeile den Namen der Klasse enthält. Im linken Teil der Karte werden nun die Verantwortlichkeiten, im rechten Teil die Partner der zu modellierenden Klasse aufgeführt.

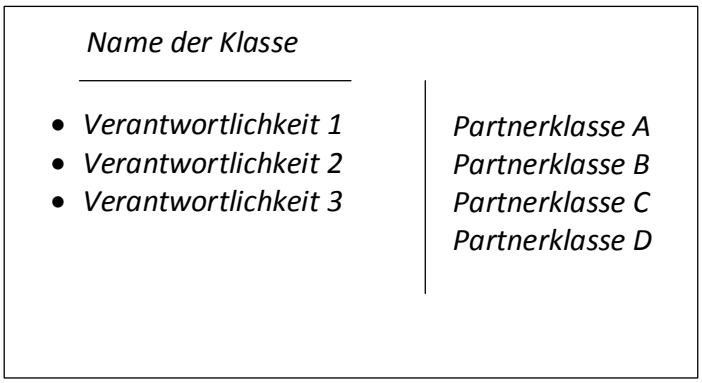


Bild 2.7: Prototyp einer CRC-Karte

Metadaten

Zusätzlich zu den Dokumenten, welche die eigentlichen Nutzdaten enthalten, werden zu jeder Quelle auch Metadaten gespeichert. Die Vorstellung der zugehörigen Implementierungsaspekte befindet sich in Ausarbeitungsabschnitt 3.2.1.

Implementierung der Datenzugriffsschicht

Der Open Data Service speichert seine Daten in einer zentralen Datenbank, welche sowohl von der Server- als auch von der Importkomponente mithilfe der Datenzugriffsschicht angesprochen wird (vergleiche 2.3.1.3). Die Beschreibung dieser Schicht ist in Abschnitt 3.2.2 des ergänzenden Forschungskapitels zu finden.

2.4.1.1 Implementierung der Serverkomponente

In diesem Teilabschnitt wird der konkrete Entwurf der in 2.3.1.1 eingeführten ODS-Serverkomponente vorgestellt. Zunächst werden die umgesetzten Routing-Klassen präsentiert, welche die HTTP-Anfragen des Clients auf die passenden Bearbeitungseinheiten verteilen. Diese, *Restlet* genannten, Bearbeitungseinheiten werden im Anschluss besprochen. Sowohl die *Router* als auch die namensgebenden *Restlets* basieren dabei auf der *Restlet*-Bibliothek. Eine Übersicht über die Server-Schnittstelle des ODS befindet sich in Abschnitt 3.2.3 des Ausarbeitungskapitels.

Routing der Anfragen

CRC-Karte 2.8 beschreibt die Aufgaben und die Interaktionspartner der *Router* im Open Data Service. Jede Klasse, die das *Router*-Interface implementiert, definiert für einen bestimmten „Application Programming Interface (API)“-Zweig die Verteilung von Anfragen auf die im Anschluss beschriebenen *Restlets*. Möchte man beispielsweise die Messwerte von *PEGELONLINE*-Stationen abfragen, so geschieht dies unter dem Zweig */ods/de/pegelonline/*. Für diese Routen weist der *PegelOnlineRouter* passende Bearbeitungs-*Restlets* zu. Möchte man OSM-Daten abfragen, so geschieht das analog dazu unter dem im *OsmRouter* definierten Teilbereich */ods/de/osm/*. Angelegt und zusammengeführt werden die einzelnen *Router* in der Klasse *ContainerRestletApp*, welche später noch genauer vorgestellt wird. Zur Erstellung der *Router* wird eine *Factory* (vergleiche Gamma et al. (1994)), die *RouterFactory*, genutzt.

Die in Diagramm 2.9 modellierte *Router*-Schnittstelle ist sehr simpel aufgebaut. Es existiert lediglich die Methode *getRoutes*, in der die Verteilung der Anfragen auf die Bearbeitungsklassen definiert wird. Diese Klassen sind gekennzeichnet durch den Typparameter T, für den im Fall des ODS stets ein *Restlet* verwendet wird.

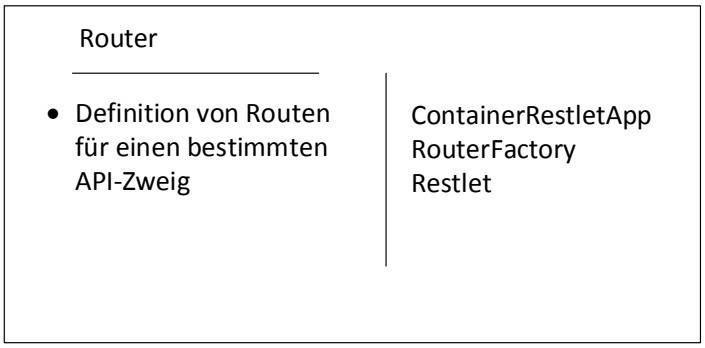


Bild 2.8: CRC-Karte der *Router*-Schnittstelle

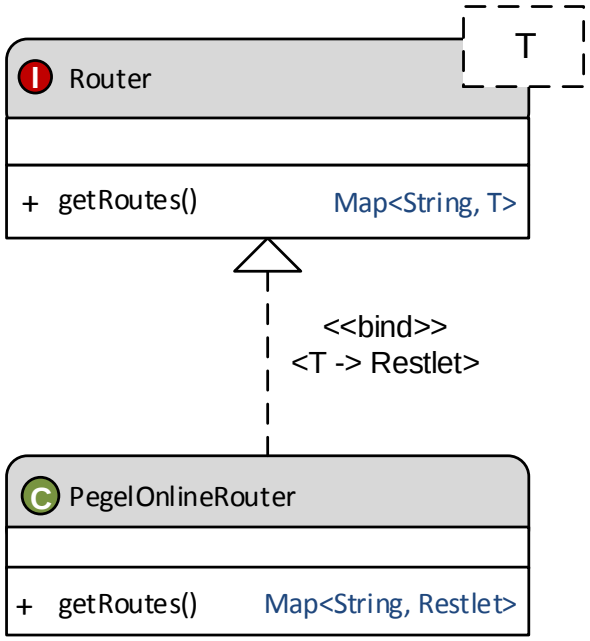


Bild 2.9: Klassendiagramm der *Router*-Schnittstelle und einer Unterklasse

Neben den quellspezifischen API-Zweigen wie */ods/de/pegelonline/* existieren auch ODS-übergreifende Zugriffswege, welche im *OdsRouter* definiert werden. Ein Beispiel dafür ist der Zugriff auf Dokumente über deren eindeutige ODS-IDs. Der Aufruf */ods/$1234* liefert das Dokument mit der Nummer 1234, wobei das Dollarzeichen hierbei auf eine ID hinweist.

Ein Spezialfall ist der als letztes erstellte *ApiRouter*. Dieser ist für die Ausgabe der Gesamtschnittstelle beim Aufruf von */api* zuständig. Dazu definiert er ein *Restlet*, das als Eingabewerte alle bislang definierten Routen und damit auch alle zugreifbaren Adressen des Servers erhält. Diese werden zusammengeführt und als Ergebnis der Anfrage zurückgegeben (siehe 3.2.3).

Behandlung der Anfragen

Die Behandlung der Client-Anfragen erfolgt in den sogenannten *Restlets*, welche in der CRC-Karte 2.10 vorgestellt werden. Diese können entweder, wie beim *ApiRouter*, direkt im betreffenden *Router* als anonyme, innere Klassen definiert werden oder bei umfangreicherer Funktionalität in richtige Klassen ausgelagert und referenziert werden. Falls die Clients Datensätze anfordern, greifen die *Restlets* dabei mittels *DbAccessor* auf die Datenbank zu.

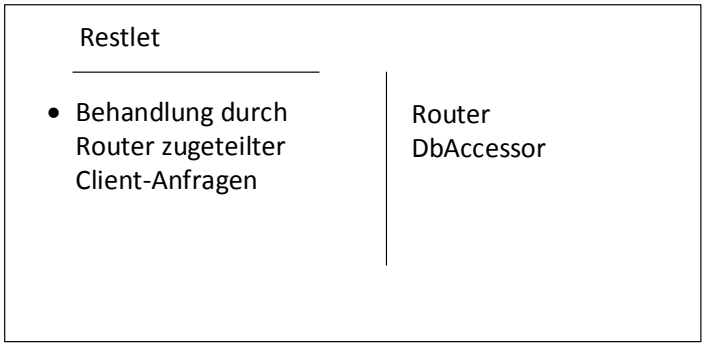


Bild 2.10: CRC-Karte der *Restlets*

Abbildung 2.11 zeigt das Klassendiagramm einer *Restlet*-Hierarchie. Jedes *Restlet* besitzt durch ihre abstrakte Oberklasse die Methode *handle*, in der die jeweilige Anfrageverarbeitung durchgeführt wird. Das ebenfalls abstrakte *BaseRestlet* dient als Grundlage aller innerhalb des ODS implementierten *Restlets*. Es beschreibt unter anderem in *onBadRequest* sowie *onInvalidMethod* die Reaktion auf ungültige Anfragen und extrahiert in *getParameter* URL-Parameter.

Das am häufigsten verwendete *Restlet* ist das *ExecuteCouchQueryRestlet*. Es übersetzt eine Client-Anfrage auf eine vordefinierte *CouchDB*-Datenbankabfrage (vergleiche 3.2.2). Dazu müssen der Name der Abfrage und der des *DesignDocuments*, in dem diese gespeichert ist, übergeben werden. Möchte man nach einem speziellen Dokumentattribut suchen (z.B. nach einer Pegelmessstation mit einem bestimmten Namen), wird zusätzlich der Name des Felds benötigt.

Weitere wichtige *Restlets* sind das *DefaultRestlet*, welches alle nicht zuordenbaren Anfragen bearbeitet, und das *AccessObjectByIdRestlet*, welche den beschriebenen globalen Dokumentzugriff über eine ID umsetzt.

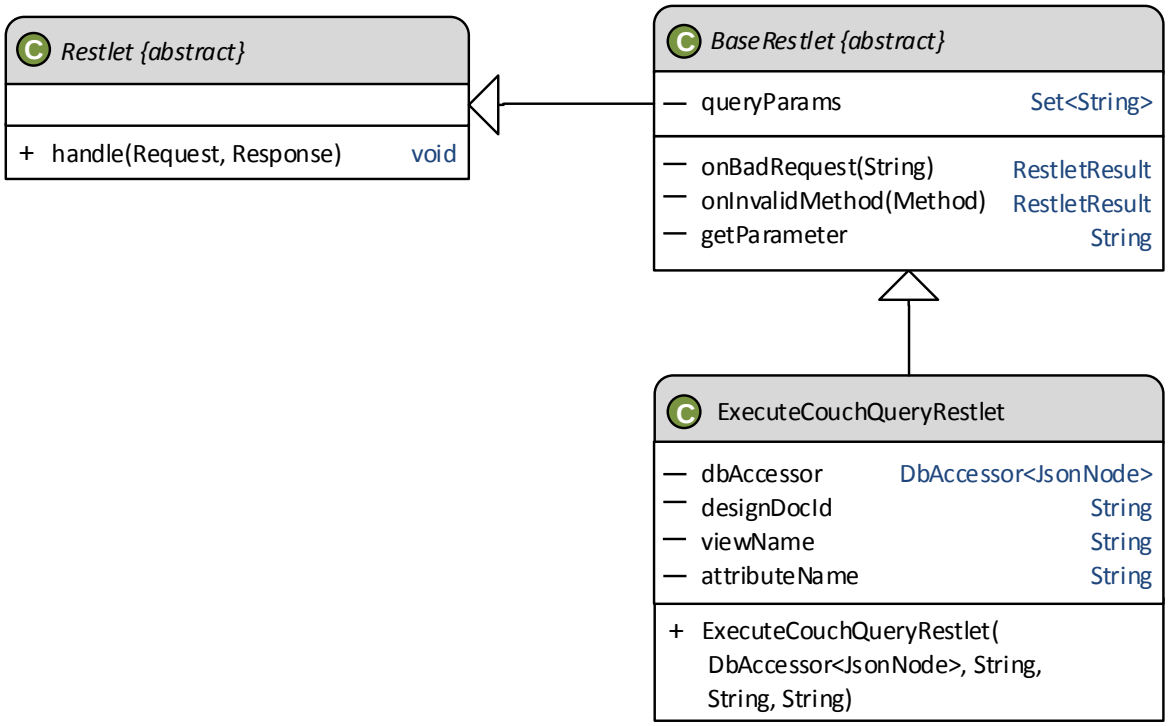


Bild 2.11: Klassendiagramm einer *Restlet*-Hierarchie

Initialisierung des Servers

Die Initialisierung des Servers erfolgt in der Klasse *ServerMain* (Abbildung 2.12). Sie erstellt die *ContainerRestletApp*, welche für das *Restlet*-Framework als Server-Applikation benötigt wird. Anschließend wird diese auf einem vordefinierten Port gestartet. Die *ContainerRestletApp* (Abbildung 2.13) ist ihrerseits zum einen für die Erstellung der Router zuständig, welche im Verlauf des Abschnitts bereits beschrieben wurden. Zum anderen initiiert sie die periodische Aktualisierung der ODS-Datenquellen, indem sie in gewissen Zeitabständen die in 2.4.1.2 vorgestellte Importkomponente aufruft.

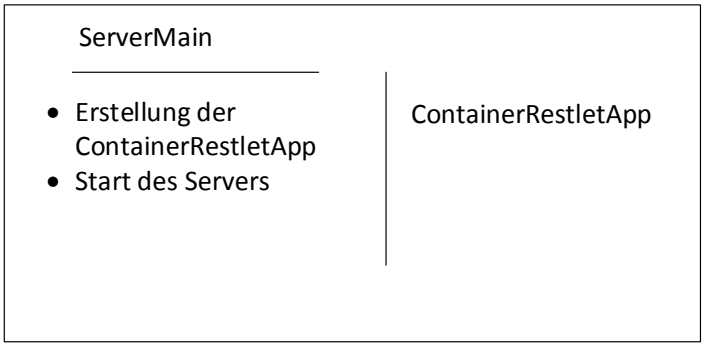


Bild 2.12: CRC-Karte der *ServerMain*-Klasse

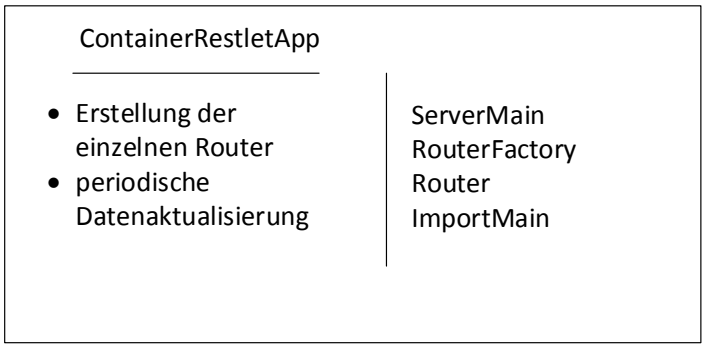


Bild 2.13: CRC-Karte der *ContainerRestletApp*

2.4.1.2 Implementierung der Importkomponente

Im folgenden Teilabschnitt wird die ODS-Importkomponente detailliert beschrieben. Sie setzt das in 2.3.1.2 erarbeitete Filterungskonzept um. Die Implementierungsdetails werden in der Reihenfolge der einzelnen Filterungsschritte vorgestellt. Abschließend wird ein Gesamtüberblick über die Abläufe innerhalb der Importkomponente gegeben. Die Einführung der Quellkonfigurationsobjekte erfolgt in Abschnitt 3.2.4 des Ausarbeitungskapitels. Eine Beschreibung der Filterketteninfrastruktur ist ebenfalls bei den Ergänzungen, in 3.2.5, zu finden.

Implementierung der Quelladapter

Den ersten Schritt jeder Filterkette stellt der Quelladapter-Filter dar. Im Feinentwurf wird dieser Filterungsschritt in zwei Teile aufgespalten. Zunächst werden die Daten mithilfe eines *Grabber* aus dem Netzwerk geladen. Anschließend werden die Daten durch Aufruf eines *Translator* in ein JSON-kompatibles Objektformat übersetzt. Da *PEGELONLINE* bereits JSON nutzt, werden die Klassen in diesem Abschnitt am aufwendigeren Beispiel von OSM eingeführt. Die OpenStreetMap-Daten basieren auf dem „*OSM XML*“-Format.

Grabber

CRC-Karte 2.14 und Klassendiagramm 2.15 beschreiben zunächst den *Grabber*. Erstellt werden die quellspezifischen *Grabber* wiederum in einer *Factory*. Da es sich bei ihnen um einen Untertyp der *Filter* handelt, erfolgt der Aufruf durch ein Objekt der Klasse *FilterChain* (vergleiche 3.2.5). Die Nutzdaten werden dann in der Methode *grabSource* heruntergeladen, wobei die notwendigen Quellinformationen aus dem *DataSource*-Objekt (vergleiche 3.2.4) bezogen werden. Der Input-Parameter T wird im *Grabber* auf Void

festgelegt, da es sich um den ersten Schritt der Filterkette handelt und daher noch keine Daten vorhanden sind. Die heruntergeladenen Daten werden im Fall des für OSM verwendeten *ResourceGrabber* als *File* weitergegeben, weswegen der Output-Parameter U diesen Typ erhält.

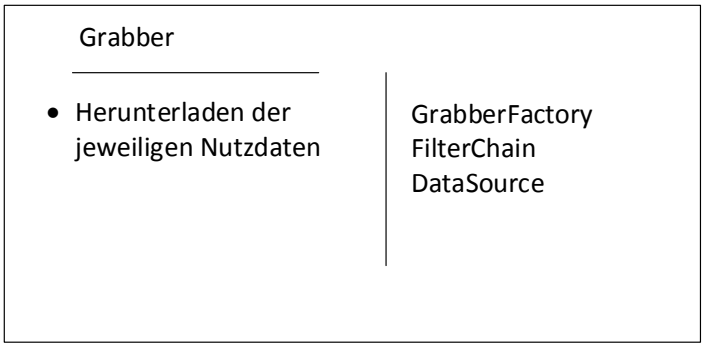


Bild 2.14: CRC-Karte des *Grabber*

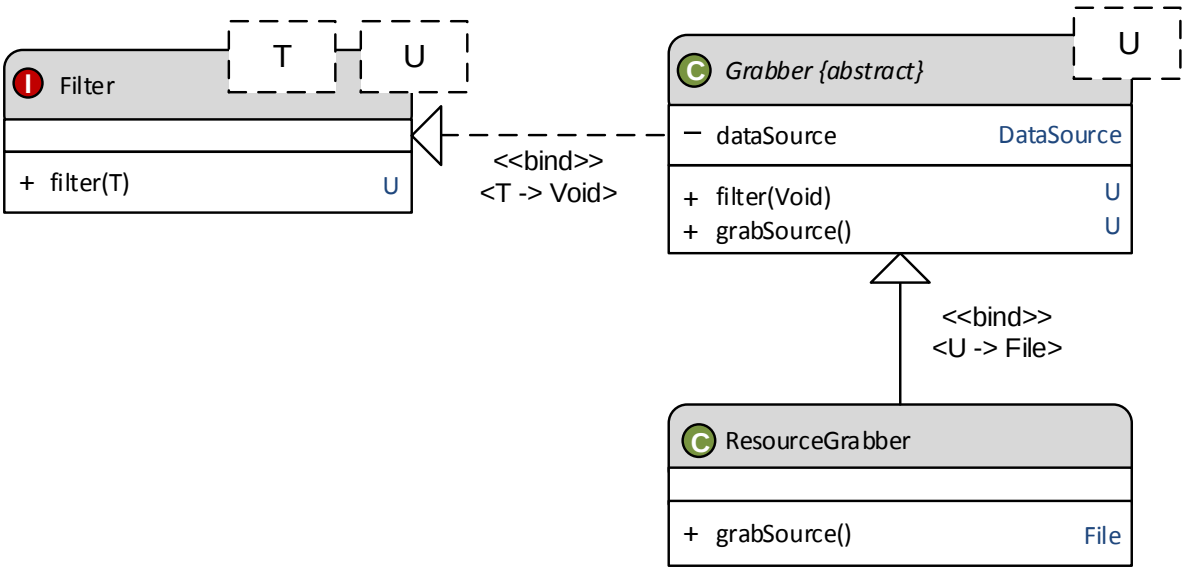


Bild 2.15: Klassendiagramm des *Grabber* und seiner Hierarchie

Translator

Die CRC-Karte 2.16 und das Klassendiagramm 2.17 modellieren nun den *Translator*. Er wird ebenfalls in einer *Factory* erstellt und, da auch er eine Spezialisierung eines

Filter darstellt, durch eine *FilterChain* aufgerufen (vergleiche 3.2.5). Die Übersetzung des Quelldatenformats in ein JSON-kompatibles Format erfolgt in der Methode *translate*.

Als Output des *Translator* dienen JSON-kompatible Objekte. Der Input ist im Fall des modellierten *OsmTranslator* vom Typ *File*. Liegen die Daten, wie bei *PEGELONLINE*, hier bereits im JSON-Format vor, ist der Typ *JsonNode*.

In der Methode *translate* nutzt der *OsmTranslator* nun zum effizienten Extrahieren der Objekte aus dem Dateiformat *OSM XML* die Bibliothek *Osmosis*[1]. Diese gibt eine Menge von OSM-Objekten (vergleiche 2.1.2.2) zurück, die in den Methoden *convertNodeToMap*, *convertWayToMap* und *convertRelationToMap* in ihren jeweiligen späteren Dokumenttyp umgewandelt werden.

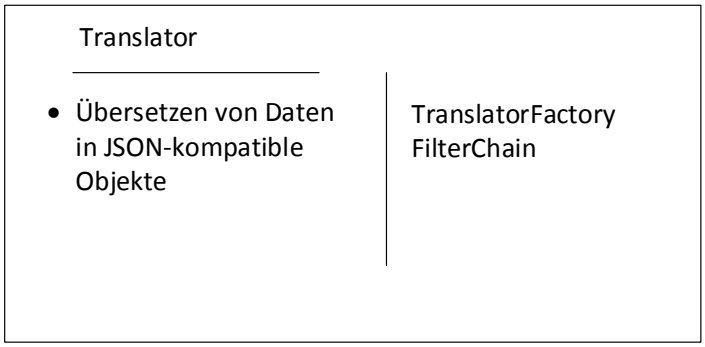


Bild 2.16: CRC-Karte des *Translator*

1 `http://wiki.openstreetmap.org/wiki/DE:Osmosis`

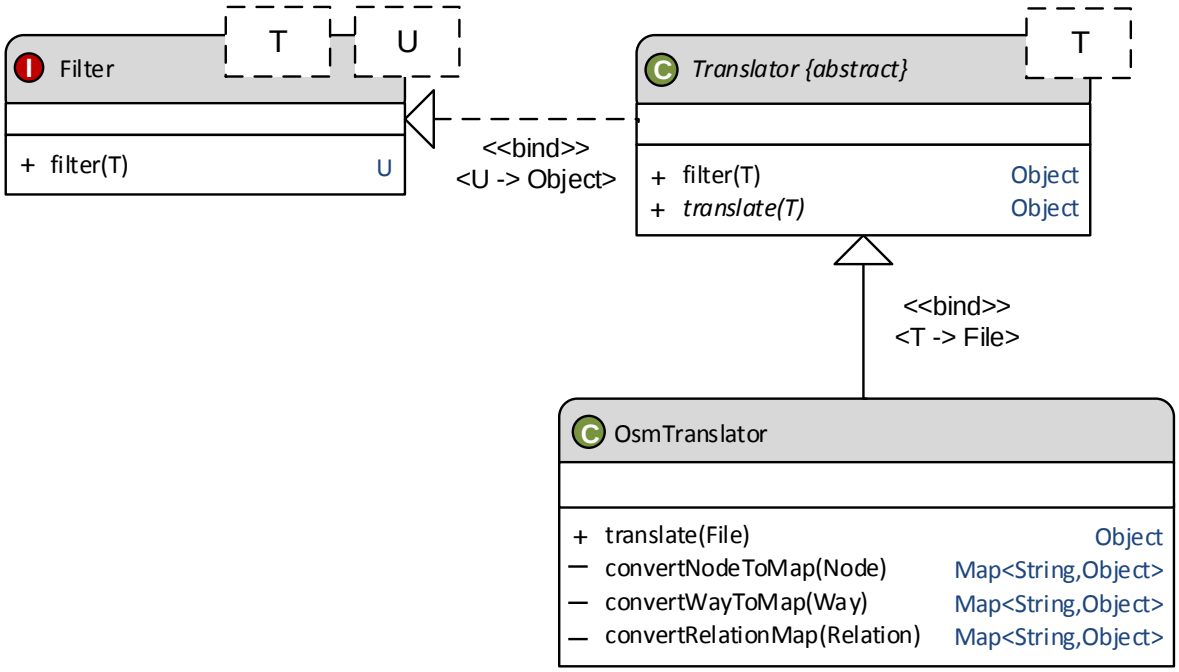


Bild 2.17: Klassendiagramm des *Translator* und seiner Hierarchie

Der Datenergänzungsfilter

Der nächste Schritt der Filterkette, die Datenergänzung, wird ebenfalls in einer auf dem *Filter*-Interface basierenden Klasse, dem *DbAdditionFilter*, umgesetzt. In seiner Implementierung der *filter*-Methode werden die durchlaufenden Dokumente um zusätzlich benötigte Felder ergänzt, die aktuell den Datentyp und den Datenqualitätsstatus innerhalb des ODS umfassen.

Der DB-Einfüge-Filter

Die importierten sowie die qualitätsverbesserten Daten werden mithilfe des DB-Einfüge-Filters in die Datenbank geschrieben. Dessen Beschreibung ist, im Rahmen der Vorstellung der Datenzugriffsschicht, in Ergänzungsabschnitt 3.2.2 zu finden.

Die Qualitätsverbesserungsfilter

Die Rohdaten der Quellen des Open Data Service können mithilfe von Qualitätsverbesserungsfiltern angepasst werden. Die Vorstellung dieser Komponenten erfolgt in Abschnitt 2.4.2.1.

Initialisierung der Importkomponente

Die Klasse *ImportMain* stellt den Einstiegspunkt in die Importkomponente dar. Die CRC-Karte 2.18 sowie das Klassendiagramm 2.19 veranschaulichen deren Zuständigkeiten respektive Architektur. In der Methode *configureAll* wird der in 3.2.4 beschriebene *ConfigurationManager* angestoßen. Da diese Konfiguration in der Regel nur einmal notwendig ist, existiert außerdem die Methode *isInitialized*, welche feststellt, ob die Initialisierung bereits stattgefunden hat. Bei jedem Aufruf der Importkomponente wird hingegen mittels *updateData* der *FilterChainManager* ausgeführt, welcher dann die konfigurierten Filterketten startet.

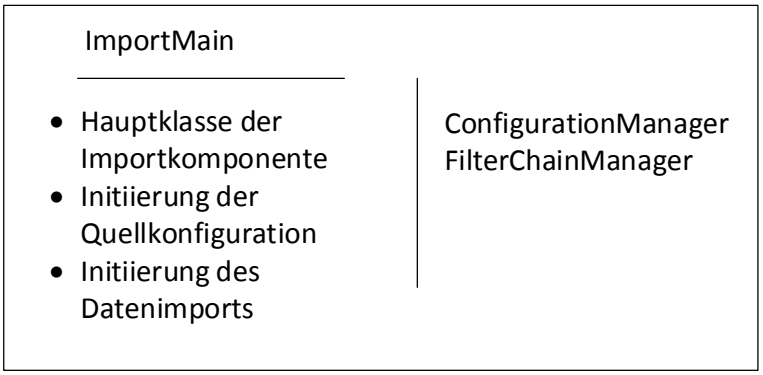


Bild 2.18: CRC-Karte der Klasse *ImportMain*

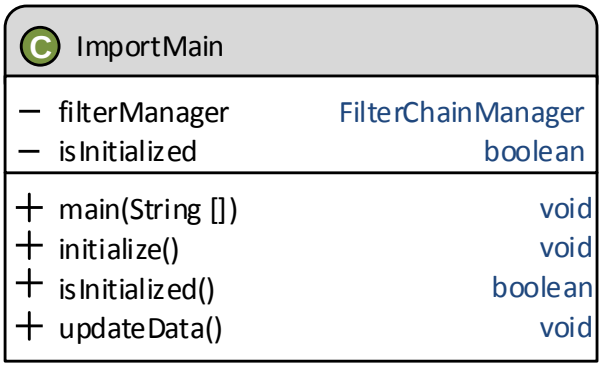


Bild 2.19: Klassendiagramm von *ImportMain*

Interaktionen innerhalb der Importkomponente

Im bisherigen Verlauf dieses Abschnittes sowie im Ergänzungskapitel 3.2 wurden die einzelnen Bestandteile der Importkomponente hauptsächlich statisch modelliert und deren Kommunikation mit direkten Interaktionspartnern beschrieben. Zum Abschluss

kann nun mithilfe des Sequenzdiagramms 2.20 ein Gesamtüberblick über die wichtigsten Abläufe innerhalb der Importkomponente gegeben werden. Die eigentliche Filterkette, welche aus den hintersten vier Akteuren besteht, ist in den bereits in 2.3.1.2 verwendeten Farben dargestellt. Die restlichen Komponenten sind grün gefärbt.

Der Ausgangspunkt des Programmes ist die Klasse *ImportMain*. Diese ruft bei der initialen Ausführung zunächst die Methode *configureAll* des *ConfigurationManager* auf. Dort werden die Quellbeschreibungen, welche unter anderem die individuellen Filterketten und die Schemata enthalten, verarbeitet und beim *FilterChainManager* mittels *register* registriert.

Nach diesem Schritt initiiert *ImportMain* die äußere Filterkette, indem sie die Methode *startFilterChains* des *FilterChainManager* aufruft. In dieser Methode wird die äußere Filterkette abgearbeitet und für jede Datenquelle wiederum deren *FilterChain*, die innere Filterkette, initiiert. Die *FilterChain* startet nun nacheinander die konfigurierten Datenfilterungsschritte. Aus Gründen der Anschaulichkeit wurde im Diagramm nur ein *FilterChain*-Objekt gezeichnet. In der konkreten Umsetzung übergibt die ursprüngliche Filterkette nach jedem Filterungschritt die Kontrolle an eine neue, um den eben durchgeführten Schritt reduzierte, Kette.

Die Datenfilter umfassen für jede Quelle des Open Data Service mindestens einen *Grabber*, einen *Translator*, einen *DbAdditionFilter* und einen *DbInsertionFilter*. Danach können bei Bedarf noch weitere Filter zur Qualitätsverbesserung und ein weiterer Einfüge-Filter zum Speichern der verbesserten Daten eingehängt werden.

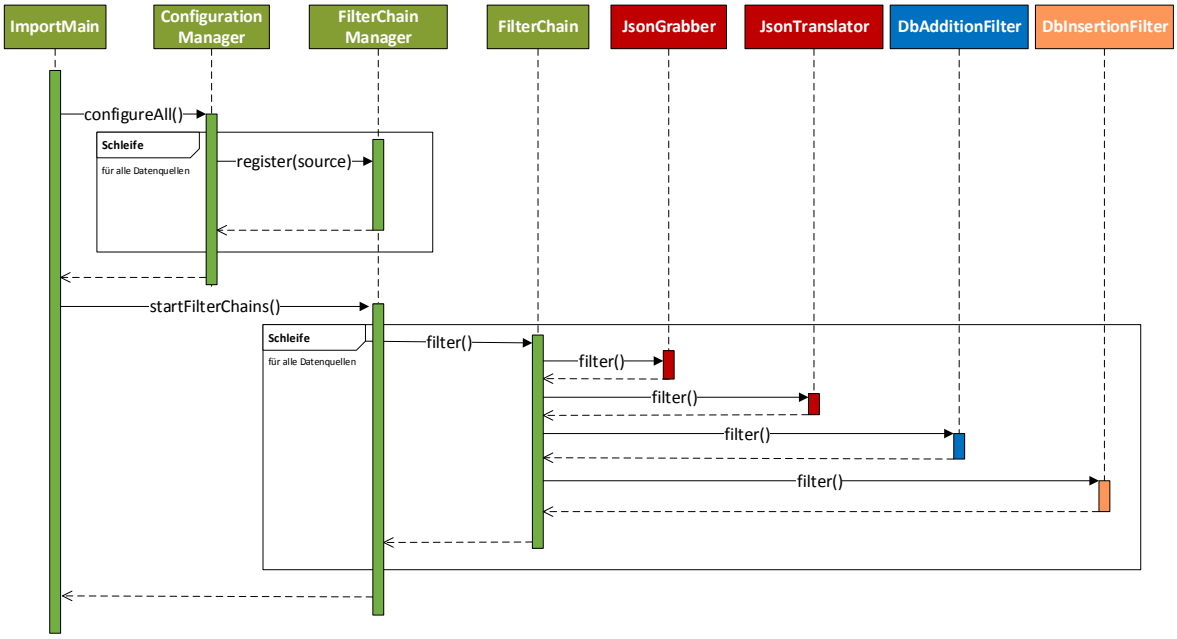


Bild 2.20: Sequenzdiagramm der Importkomponente

2.4.2 Implementierung der Datenqualitätsverbesserung

In diesem Abschnitt werden die in 2.3.2 vorgestellten Qualitätsverbesserungsmaßnahmen im Detail präsentiert. Zunächst erfolgt in 2.4.2.1 die Beschreibung der Datenqualitätsfilter. Danach werden in 2.4.2.2 die umgesetzten *Value Objects* behandelt.

2.4.2.1 Implementierung der Qualitätsverbesserungsfilter

CombineFilter

Die CRC-Karte 2.21 und das Klassendiagramm 2.22 beschreiben den *CombineFilter*. Seine Aufgabe ist es, zusammengehörige Felder einer flachen Hierarchie in einem neuen Container-Objekt zu vereinigen. Damit schafft er die Basis für die anschließend in 2.4.2.2 vorgestellten *Value Objects*. Die *CombineFilter* werden für jede Quelle in ihrer jeweiligen Konfigurationsklasse (vergleiche 3.2.4) definiert. Für jedes neu zu generierende Objekt wird dabei ein eigener *CombineFilter* erstellt. Der Aufruf erfolgt wie bei jedem *Filter* in einer *FilterChain*.

Input- und Output-Parameter der Qualitätsverbesserungsfilter können beliebige Objekte sein. Der Konstruktor des *CombineFilter* bekommt zwei Strukturen übergeben. Diese beschreiben sowohl die Ursprungsstruktur als auch die Zielstruktur des Dokuments. Bei den Strukturen handelt es sich um Ausschnitte der Dokumentschemata (vergleiche 3.2.1),

die auf die betreffenden Felder und deren Container reduziert sind. So kann die richtige Hierarchietiefe zunächst festgelegt und später wieder ermittelt werden, da Feldernamen nur auf ihrer jeweiligen Objektebene eindeutig sind. In der Methode *traverseSchema* wird anhand der Ursprungsstruktur zu den zusammenzuführenden Feldern navigiert, um diese vorübergehend aus dem Dokument zu entfernen. *insertCombinedObject* fügt das neue Container-Objekt schließlich wieder in das Dokument ein.

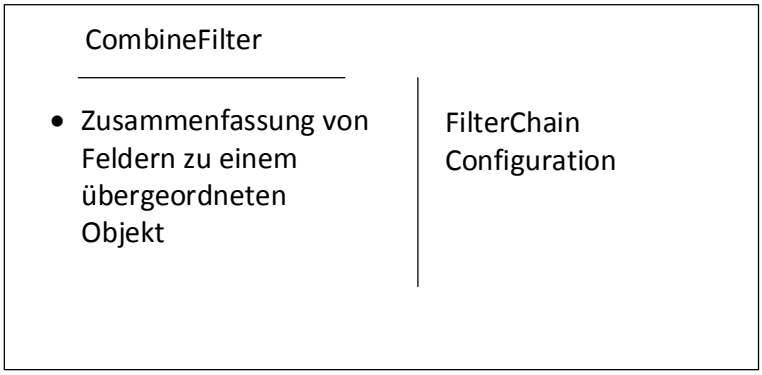


Bild 2.21: CRC-Karte des *CombineFilter*

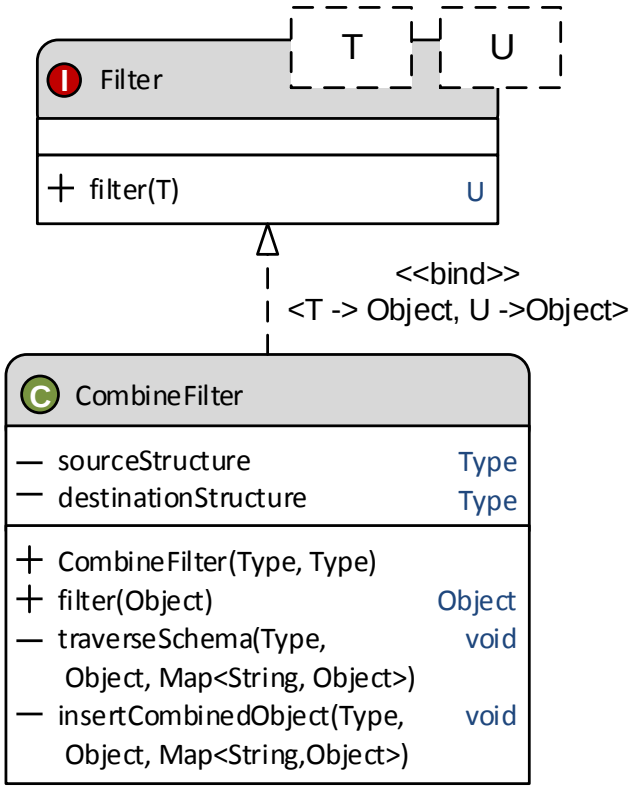


Bild 2.22: Klassendiagramm des *CombineFilter*

RenameFilter

Der in den Abbildungen 2.23 und 2.24 modellierte *RenameFilter* ist ähnlich wie der *CombineFilter* aufgebaut. Er wird in der Quellkonfiguration definiert und von einer *FilterChain* initiiert. Seine Aufgabe ist es, Feldernamen umzubenennen und so eine einheitliche Bezeichnung gleicher Datentypen innerhalb des Open Data Service zu erreichen. In der Methode *traverseSchema* wird das zu bearbeitende, diesmal einzelne, Feld aus dem Dokument extrahiert, um anschließend in *insertRenamedObject* unter seinem modifizierten Namen neu eingefügt zu werden. Der *RenameFilter* kann ebenfalls mit beliebigen Objekttypen parametrisiert werden.

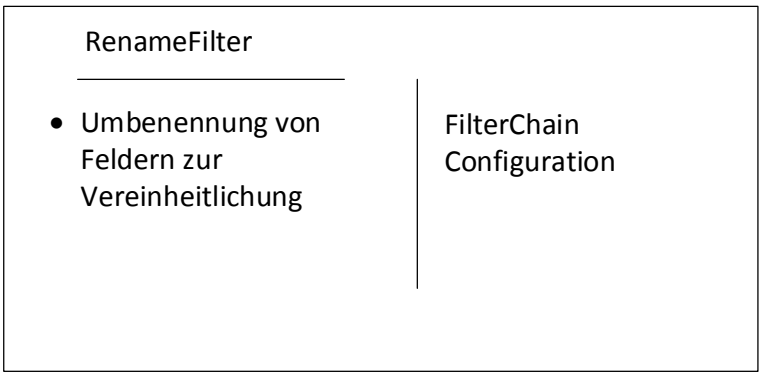


Bild 2.23: CRC-Karte des *RenameFilter*

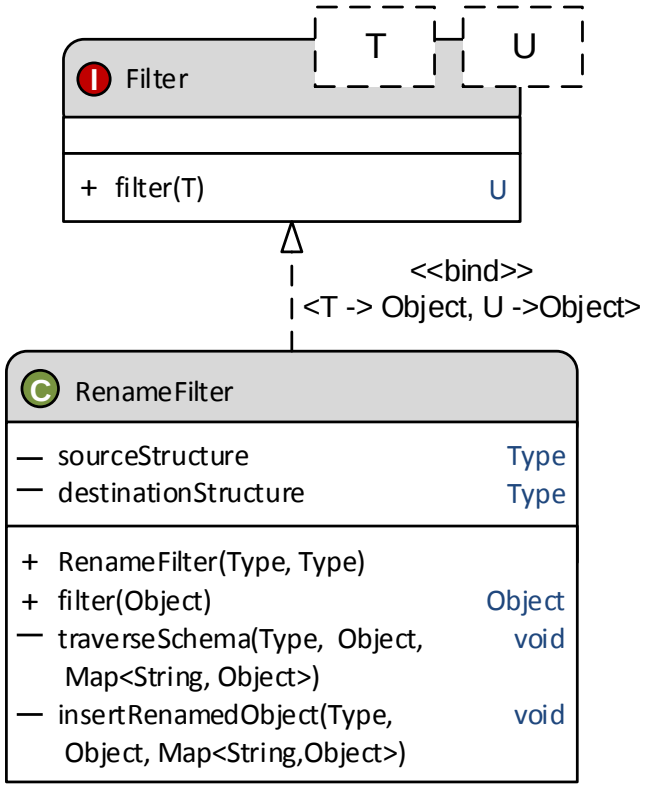


Bild 2.24: Klassendiagramm des *RenameFilter*

2.4.2.2 Implementierung der *Value Objects*

Konkrete *Value Objects*

Ein Beispiel für einen konkret implementierten *Value Object Type* ist die in Abbildung 2.25 modellierte Klasse *Coordinate*. Sie baut auf dem in 2.1.1.2 beschriebenen *JValue*-Framework auf. Koordinaten enthalten als Felder einen Breitengrad und einen Längengrad. Zusätzlich dazu werden zwei Typ-Felder erstellt, die den Wertebereich der beiden Fließkommazahlen weiter einschränken. Innerhalb eines *JValue-NumberType* können hierfür *Ranges* definiert werden. Im Fall des Breitengrads werden die gültigen Werte auf -90 bis +90 beschränkt. Analog dazu muss ein Längengrad immer im Bereich -180 bis 180 liegen (vergleiche 2.2.2). Im Konstruktor der *Coordinate*-Klasse werden die zwei übergebenen Argumente nun nicht einfach gespeichert, sondern zunächst in der *validate*-Methode des *NumberType* überprüft. Liegt einer der Werte außerhalb seines definierten Bereichs, wird eine Exception geworfen und die Objekterstellung abgebrochen.

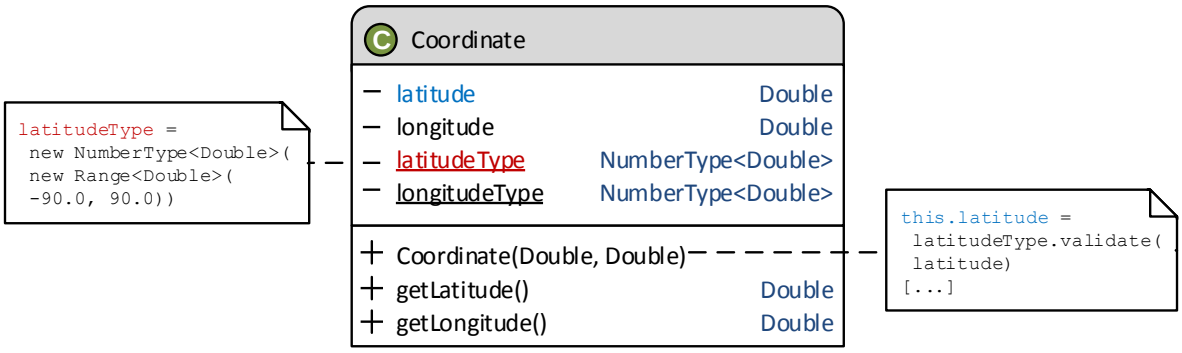


Bild 2.25: Klassendiagramm von *Coordinate*

Dynamische *Value Objects*

Zusätzlich zu konkreten werden im Rahmen des ODS hauptsächlich dynamische *Value Objects* definiert. Ein anschauliches Beispiel dafür ist der in Listing 2.1 dargestellte *MeasurementTrendType* von *PEGELONLINE*. Er beschreibt die Tendenz eines Messwerts (z.B. eines Pegelstands oder einer Temperatur) mithilfe von vier fest definierten Zeichenketten und kann daher als ein im Anschluss beschriebener *EnumType* umgesetzt werden. -1 weist auf das Sinken, 1 auf das Steigen und 0 auf das Stagnieren des Messwerts hin. Der Wert -999 dient als Fehlerzustand [1]. Um nur die vier Zustände zuzulassen, werden diese jeweils als *JValue-ExactValueRestriction* kodiert und bei der Erstellung des *EnumType* verodert. Der entstandene Wertetyp kann durch den Aufruf der Methode *isValidInstance* nun die Gültigkeit des Trends eines *PEGELONLINE*-Messwerts prüfen.

1 `http://www.pegelonline.wsv.de/webservice/dokuRestapi`

```
1     private static EnumType createMeasurementTrendType() {
2         ExactValueRestriction<String> a =
3            new ExactValueRestriction<String>("-1");
4         ExactValueRestriction<String> b =
5            new ExactValueRestriction<String>("0");
6         ExactValueRestriction<String> c =
7            new ExactValueRestriction<String>("1");
8         ExactValueRestriction<String> d =
9            new ExactValueRestriction<String>("-999");
10        EnumType trendType = new EnumType(a.or(b).or(c).or(d));
11        return trendType;
12    }
13    [...]
14    createMeasurementTrendType().isValidInstance(
15       document.get("trend"));
```

Listing 2.1: Implementierung des dynamischen Wertetyps *MeasurementTrendType*

EnumType

JValue wurde im Rahmen dieser Arbeit um den *EnumType* (Abbildung 2.26) ergänzt. Die abstrakte Oberklasse aller *JValue*-Typen, *ValueType*, definiert drei Methoden sowie den Typparameter T. *isValidInstance* führt die eigentliche Validierung des Datentyps durch und gibt das Ergebnis als booleschen Wert zurück. Die Methode *assertIsValidInstance* ruft diese Validierungsmethode auf und wirft bei negativem Rückgabewert eine *IllegalArgumentException*. *validate* verwendet wiederum letztere Methode und gibt bei erfolgreicher Validierung den Eingabewert zurück, was die Benutzung innerhalb einer Zuweisungsoperation ermöglicht (vergleiche Bild 2.25).

Der davon abgeleitete *SimpleValueType* repräsentiert in *JValue* vorwiegend primitive Datentypen. Er erhält im Konstruktor eine *Restriction*, deren Einhaltung in der Implementierung der Methode *isValidInstance* geprüft wird.

Ein *String*-parametrisierter *SimpleValueType* dient allerdings auch als Grundlage für den neu konstruierten *EnumType*. Dieser wird entweder durch die Übergabe einer bereits vorhandenen „Oder-*Restriction*“ oder durch die Übergabe einer Sammlung erlaubter Zeichenketten erstellt. Im zweiten Fall wird die Veroderung der Zeichenketten in der Methode *createRestriction* durchgeführt.

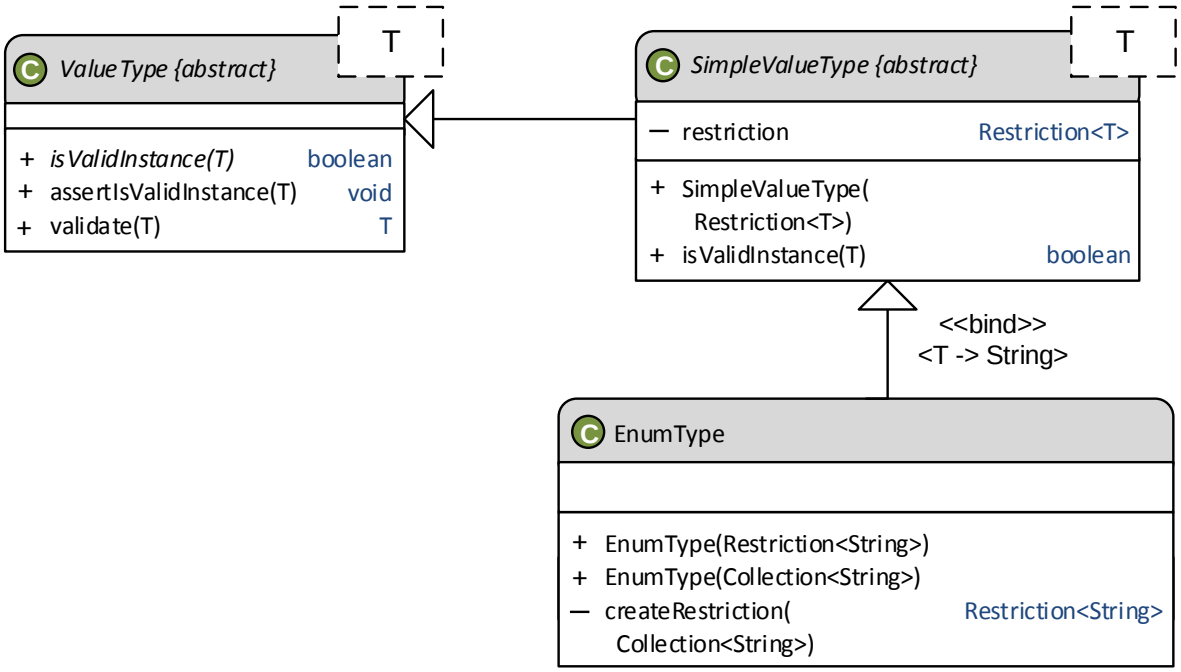


Bild 2.26: Klassenhierarchie des *EnumType*

Einsatz der *Value Objects*

Die in Abbildung 2.27 modellierte Klasse *PegelOnlineAssurance* definiert dynamische *Value Objects* für die Datensätze aus *PEGELONLINE*. Die Klasse wird ebenfalls als *Filter* implementiert und innerhalb der *PegelOnlineConfiguration* zur Kette hinzugefügt. Input- und Outputparameter werden als beliebige Objekte definiert, wobei zu beachten ist, dass sich der Inhalt der Datensätze innerhalb der Klasse nicht verändert. Die Methoden *createMeasurementTrendType*, *createWaterLevelType*, *createTemperatureType* etc. definieren die dynamischen *Value Object Types*. In der *filter*-Methode werden diese dann, zusammen mit Instanzen der konkreten *Value Objects*, zur Validierung der Daten verwendet.

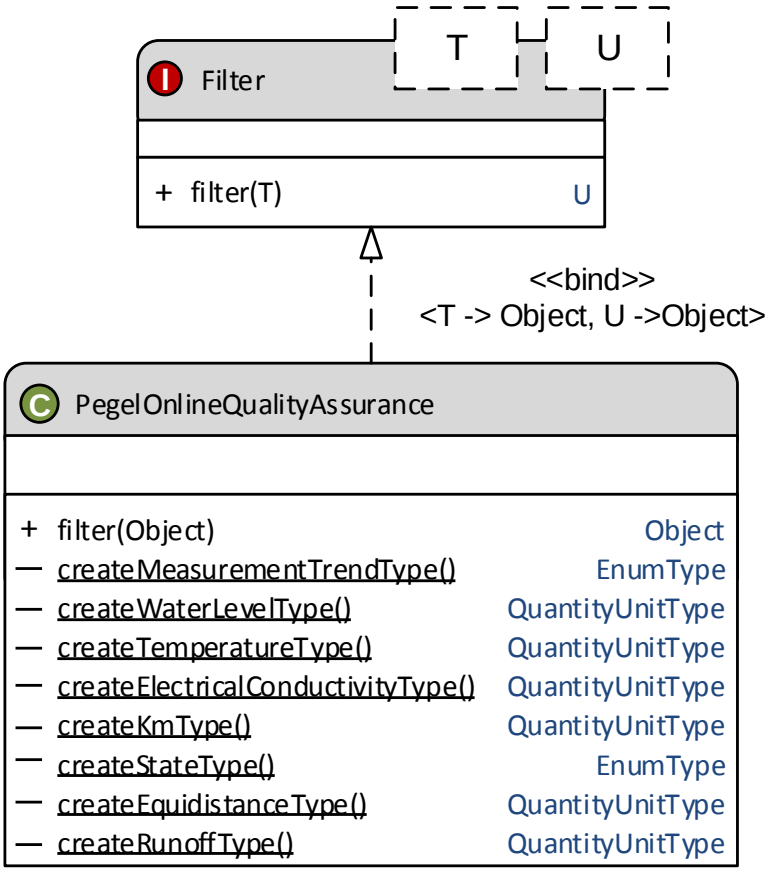


Bild 2.27: Klassendiagramm der *PegelOnlineQualityAssurance*

2.5 Diskussion und Ausblick

Der folgende Abschnitt setzt sich mit den Ergebnissen der Arbeit kritisch auseinander und gibt einen Ausblick auf die Zukunft des Open Data Service. Zunächst wird in 2.5.1 analysiert, ob und inwieweit die in 2.2.1 sowie 3.1 definierten Anforderungen an den ODS und an die Datenqualitätsverbesserung erfüllt worden sind. Darauf basierend erfolgt in 2.5.2 eine Beschreibung ausstehender Arbeiten, die bisher noch nicht umgesetzt wurden

2.5.1 Diskussion der Ergebnisse

Der vorgestellte Open Data Service hat das Ziel erreicht, Gewässerdaten zentral zu speichern und mithilfe einer auf REST basierenden Schnittstelle benutzerfreundlich zur Verfügung zu stellen. Insbesondere können Daten aus verschiedenen Quellformaten extrahiert, eingefügt und angeboten werden. Die importierten Datenquellen umfassen *PEGELONLINE*, *Pegelportal Mecklenburg-Vorpommern* und *OpenStreetMap*. Die im ODS verwendete dokumentenorientierte Datenbank *CouchDB* speichert die Daten im übersichtlichen JSON-Format, welches sowohl mit Datensätzen im Internet als auch mit Objektrepräsentationen in Programmiersprachen eng verwandt ist. Für *CouchDB* wurde eine Reihe von vordefinierten Datenbankabfragen erfolgreich implementiert. Dadurch können die Benutzeranfragen effizient bearbeitet werden.

Die Beispielanwendung des ODS wurde auf einem Server der Universität Erlangen-Nürnberg installiert. Die Datenquellen werden regelmäßig aktualisiert, wobei jeweils die neuesten Versionen der Dokumente übernommen werden, ohne dabei auf konkrete Änderungen zu prüfen. In Update-Phasen ist dadurch die durchgängige Erreichbarkeit nicht gewährleistet. Des Weiteren ändern sich nach der Aktualisierung die IDs der Datensätze, wodurch beim globalen ID-Zugriff die REST-Eigenschaften nicht erfüllt sind.

Gerade bei der Verarbeitung großer geografischer Datensätze aus *OpenStreetMap* gerät der Open Data Service an seine Grenzen. Das Einlesen und das Übersetzen der Quelldaten aus dem XML-Format in ein *CouchDB*-kompatibles Format dauert bereits bei recht kleinen Gebieten sehr lange. Eine sinnvolle Nutzung der Kartendaten für beispielsweise eine Pegelstands-App müsste mindestens einen deutschland- oder mitteleuropaweiten Datensatz verarbeiten können, was im Rahmen des ODS bisher nicht möglich ist. Um ein effizienteres Parsen des speziellen *OSM XML*-Formats zu ermöglichen, wurde bereits die darauf optimierte Bibliothek *Osmosis* verwendet, was die Verarbeitungsgeschwindigkeit nur leicht erhöhen konnte. Neben dem Einlesen und Umwandeln dauert jedoch auch das Lesen und Schreiben der Datenbankdokumente lange.

Daher gilt es in Zukunft zu evaluieren, ob das Anbieten von OSM-Daten innerhalb des ODS und dessen dokumentenorientierter Datenbank überhaupt realistisch ist. Es sollte insbesondere analysiert werden, welche Komponenten der Filterkette den Flaschenhals im Open Data Service darstellen.

Unter anderem das eben erwähnte Problem hat dazu geführt, dass die Daten innerhalb des ODS bislang nur experimentell verknüpft werden konnten. Eine durchgehende Abstrahierung von beispielsweise Pegelmessstationen verschiedener Datenanbieter fand nicht statt. Das Zusammenführen der Messwerte muss also immer noch von den Clients ausgeführt werden. Aus diesem Grund sind die dynamischen *Value Objects* ebenfalls nur auf einzelne Quellen beschränkt und abstrahieren keine allgemeinen Datentypen. Auch die geplante Anreicherung von Pegelmessstationsdaten mit geografischen Daten aus OSM wurde noch nicht durchgeführt.

Der Open Data Service wurde im Laufe dieser Arbeit zusammen mit einem kleinen Team von Grund auf konzipiert und umgesetzt. Da dieses Projekt das Ziel hat, langfristig weiterentwickelt zu werden und wenn möglich Partner in der Wirtschaft zu gewinnen, wurde viel Zeit mit dem Entwurf und der Optimierung der Architektur verbracht. Daher konzentrierte sich ein Großteil dieser Arbeit auf die saubere Gestaltung des Grundsystems. Die Maßnahmen zur Datenqualitätsverbesserung wurden nicht im ursprünglich geplanten Ausmaß durchgeführt. Umgesetzt wurden zwei Qualitätsverbesserungsfilter, welche Attribute zusammenführen und umbenennen können. Außerdem umfassen die Verbesserungen eine Reihe von dynamischen *Value Object Types* für *PEGELONLINE* sowie den konkreten Wertetyp *Coordinate*. Hier gilt es in Zukunft an die Forschung anzuschließen und den Umfang und die Reichweite der Verbesserungen zu erhöhen.

2.5.2 Ausstehende Arbeiten

Evaluation der Datenqualitätsverbesserungen

Im Laufe dieser Arbeit wurde mit verschiedenen Mitteln versucht, die Datenqualität im Open Data Service zu erhöhen. Aufgrund der geringen Nutzerzahl war es bislang jedoch nicht möglich, die Auswirkungen dieser Maßnahmen auf die Benutzerzufriedenheit zu bestimmen. Es wäre im Interesse der Entwickler, die Datenqualitätsverbesserungen im Open Data Service in Zukunft zu evaluieren, um weitere Verfahren noch zielgerichteter durchführen zu können. Dies ist beispielsweise interessant für die Frage, ob Rohdaten weiterhin zur Verfügung gestellt werden sollen oder ob die aufbereiteten Daten ausschließlich Vorteile mit sich bringen.

Mögliche Migration des ODS auf eine andere Datenbank

Wie bereits in 2.3 beschrieben, hat die dokumentenorientierte Datenbank *CouchDB* nicht nur Vorteile. Zum einen ist die Verknüpfung zusammenhängender Objekte im Gegensatz zu relationalen Datenbanken relativ aufwendig. Zum anderen ist die Möglichkeit, parametrisierte Datenbankanfragen zu programmieren, in *CouchDB* aufgrund des angewandten *MapReduce*-Prinzips begrenzt. Daher gilt es in Zukunft zu prüfen, ob ein Wechsel zu anderen dokumentenorientierten Datenbanken wie beispielsweise *MongoDB*[1] oder sogar zu relationalen Datenbanken die Programmierung erleichtern beziehungsweise die Geschwindigkeit der Anfrageverarbeitung im *Open Data Service* erhöhen könnte.

Dynamische Anbindung neuer Datenquellen durch Clients

Der Open Data Service sammelt Daten vieler verschiedener Datenquellen. Es besteht im konkreten Anwendungsbereich das Problem, dass jedes Bundesland eigene Gewässerdaten auf ihren jeweiligen Internetauftritten veröffentlicht. Einige wenige Seiten bieten JSON an, andere veröffentlichen Tabellen und manche Bundesländer stellen die Daten lediglich in Grafiken dar. Aus diesen Grund müssen für fast jede neu einzubindende Quelle Plugins programmiert und konfiguriert werden.

Um dies zu erleichtern, soll in Zukunft eine dynamische Anbindungskomponente für neue, homogene Datenquellen entwickelt werden. Möchte ein Client (oder auch der Host des ODS) beispielsweise neue Tabellendaten verwenden, so sollte er in Zukunft nur noch die Web-Adresse und einige weitere Konfigurationsdaten wie beispielsweise das Tabellenschema spezifizieren müssen.

Eine komplett generische Anbindung neuer Quellen wird dabei allerdings nicht möglich. Die Daten müssen stets so weit abstrahierbar sein, dass ihre Struktur nur in wenigen Parametern voneinander abweicht und die darunter liegenden Software-Schichten die Informationen nach dem gleichen Muster extrahieren können.

Eigene URL-Routing-Komponente

Wie in Abschnitt 2.3.1.1 beschrieben, wird für die komfortable Anfrageverarbeitung im Open Data Service die Bibliothek *Restlet* verwendet. Dies hat jedoch nicht nur Vorteile. Es ist damit zum Beispiel nicht trivial umsetzbar, das Verhalten für URLs unbekannter Länge zu spezifizieren.

1 `http://www.mongodb.org/`

Der ODS bietet eine experimentelle Möglichkeit, sich beliebig tiefe Ausschnitte eines Datensatzes liefern zu lassen. Handelt es sich bei dem Datensatz um eine *PEGELONLINE*-Messstation, können beispielsweise entweder das ganze Dokument, nur dessen verbessertes Feld *coordinate* oder wiederum lediglich dessen Feld *longitude* direkt abgefragt werden. Ein dynamisches Routen des Dokumentes ist jedoch mit *Restlet* nicht möglich. Stattdessen muss dieses Verhalten bereits jetzt manuell behandelt werden. Daher soll *Restlet* in Zukunft durch eine mächtigere, selbst entwickelte, Routing-Komponente ersetzt werden.

Authentifizierung von Clients

Der Ursprungsgedanke des Open Data Service ist es, freie Datenquellen im Internet zusammenzuführen und auf komfortable Art und Weise neuen Nutzern anzubieten. Aus diesem Grund war es bislang nicht nötig, den Zugang zu den Daten zu beschränken. Jedoch könnten zukünftige Nutzungsszenarien des ODS auch die Bereitstellung von Informationen einschließen, die nicht für die Allgemeinheit bestimmt sind. So könnten beispielsweise sowohl sensible, personenbezogene Daten als auch Betriebsgeheimnisse innerhalb einer Firma als Grundlage dienen. Daher wäre es sinnvoll, die Möglichkeit zur Verfügung zu stellen, Clients identifizieren sowie authentifizieren zu können. Daraufhin kann den Nutzern der Zugang zu einer entsprechenden Menge von Dokumenten gewährt werden.

2.6 Zusammenfassung

Im Laufe dieser Arbeit wurde die Entwicklung des Open Data Service und der darauf angewandten Qualitätsverbesserungsoperationen beschrieben. Die Einleitung (1) stellte zu Beginn die Notwendigkeit von Maßnahmen zur Datenqualitätsverbesserung im heutigen Informationszeitalter dar. Des Weiteren wurde auf die Problematik der dezentralen und uneinheitlichen Veröffentlichung von Gewässerdaten in Deutschland hingewiesen. Als Ziel der Arbeit wurde zum einen definiert, den Open Data Service als Beispielanwendung für das Zusammenführen und Anbieten offener Gewässerdaten zu entwickeln. Zum anderen sollte sich im Speziellen um die Datenqualitätsverbesserung mithilfe von simplen Qualitätsverbesserungsfiltern und *Value Objects* gekümmert werden.

Abschnitt 2.1 behandelte verwandte Arbeiten im Forschungsbereich. Der Begriff der Datenqualität wurde eingeführt und Beispiele für Verbesserungsmöglichkeiten dieser besprochen. Insbesondere erfolgte dabei die Beschreibung des *Value Object Pattern*. Im Anschluss daran wurden das REST-Paradigma und einige Web Services, die auf diesem basieren, thematisiert.

In Abschnitt 2.2 ging es um die Forschungsfrage. Hierzu wurden die funktionalen Anforderungen an den ODS und an die Datenqualitätssicherung beschrieben.

Danach erfolgte in 2.3 die Präsentation des Forschungsansatzes und damit des Grobkonzepts der Arbeit. Hierbei wurde die Trennung des ODS in Server- und Importkomponente sowie deren jeweilige Architektur vorgestellt. Die Importkomponente sollte insbesondere aus einer Reihe von einzelnen Bearbeitungsschritten (genannt Filter) bestehen, die unter anderem das Herunterladen, Ergänzen, Verbessern und Abspeichern der Daten übernehmen. Außerdem wurde ein Gesamtüberblick über das geplante System gegeben. Für die Qualitätssicherung wurden Kombinations- sowie Umbenennungsfilter entworfen und die Trennung der umzusetzenden *Value Objects* in konkrete und dynamische Varianten begründet. Nur wichtige, wiederverwendbare Datentypen sollten konkret implementiert werden.

Das Ergebnis der Arbeit, die prototypische Implementierung des Open Data Service und der Datenqualitätsverbesserungen, war Thema des Abschnitts 2.4. Hierfür wurden die Einzelkomponenten und das Zusammenspiel innerhalb dieser detailliert modelliert. Die Serverkomponente ist aufgeteilt in Routing- und Bearbeitungsklassen, wobei erstere die Client-Anfragen an die richtigen Bearbeitungseinheiten delegieren. Auf der anderen Seite durchläuft die Importkomponente eine Filterkette, deren Elemente eine gemeinsame Schnittstelle implementieren, und darin einen bestimmten Bearbeitungsschritt

durchführen. Des Weiteren behandelte das Kapitel die Umsetzung der geplanten Datenqualitätsfilter und die Infrastruktur zur Definition von konkreten sowie dynamischen *Value Objects*.

Abschließend wurden in Abschnitt 2.5 die Ergebnisse diskutiert und ein Ausblick auf zukünftige Arbeiten gegeben. Es wurde festgestellt, dass der Open Data Service als Datenzusammenführungs- und Datenbereitstellungsplattform grundsätzlich geeignet ist, jedoch auch Nachteile mit sich bringt. Gerade die dokumentenorientierte Datenbank kann zwar für einen hohen Nutzungskomfort sorgen, wird aber bisher insbesondere bei großen OSM-Datensätzen, die in viele Einzelteile aufgeteilt werden müssen, zu ineffizient verwendet. Es wurde außerdem festgestellt, dass die präsentierten Qualitätsverbesserungsmaßnahmen einen ersten Schritt zur Erfüllung möglichst vieler Datenqualitätskriterien im Open Data Service darstellen. Nur durch Weiterführung der Forschung kann jedoch die stetig steigende Menge an Informationen im Internet effizient und gewinnbringend genutzt werden.

3 Ausarbeitung der Forschung

Dieses Kapitel ergänzt den Hauptteil der Arbeit um weiterführende Betrachtungen einiger Themengebiete. In 3.1 werden, zusätzlich zu den in 2.2 definierten funktionalen, die nicht-funktionalen Anforderungen an den Open Data Service vorgestellt. 3.2 erweitert außerdem den Implementierungsabschnitt 2.4 um zusätzliche Klassen- und Funktionalitätsbeschreibungen.

3.1 Ergänzungen zur Forschungsfrage

In Abschnitt 2.2 wurden die funktionalen Anforderungen an den Open Data Service thematisiert. Zusätzlich dazu existieren jedoch auch einige nicht-funktionale Anforderungen, welche nun vorgestellt werden.

3.1.1 Nicht-funktionale Anforderungen an den Open Data Service

Durchgängige Erreichbarkeit

Der Open Data Service sollte auf einem Server im World Wide Web (WWW) installiert werden und durchgängig für Anfragen der Clients erreichbar sein. Auch während eines Updates sollten zumindest die gültigen alten Daten verfügbar sind.

Speicherung von Backups alter Daten

Quellen im Internet bieten häufig nur aktuelle oder auf einen bestimmten Zeitraum begrenzte Daten an. Durch die regelmäßigen Updates des ODS können Daten jedoch auch über längere Zeit, unabhängig von externen Archiven, gespeichert und zur Verfügung gestellt werden.

Effiziente Bearbeitung von Client-Anfragen

Client-Anfragen können mit wachsender Anzahl der vorhandenen Datensätze einen erheblichen Rechenaufwand erfordern. Es müssen stets erträglich lange Antwortzeiten des Servers gewährleistet werden. Dies soll einerseits durch die Wahl einer sinnvollen Datenbanktechnologie (wie bereits in 2.2.1 beschrieben) geschehen. Andererseits sollen auch nur Anfragen angeboten werden, die eine gewisse Komplexität nicht überschreiten. Möchte der Client weiterführende Auswertungen der Daten durchführen, so kann dies nach Abfrage einer größeren, nur begrenzt gefilterten Datenmenge lokal geschehen.

3.2 Ergänzungen zur Implementierung

In diesem Abschnitt wird der Software-Entwurf (2.4) des Open Data Service weiter ergänzt. 3.2.1 stellt die ODS-Typhierarchie zur Definition von Schemata sowie zusätzliche Metadatenobjekte vor. In 3.2.2 wird die von Server- und Importkomponente genutzte Datenzugriffsschicht thematisiert. Eine Beschreibung der ODS-Server-Schnittstelle erfolgt in 3.2.3. Der Teilabschnitt 3.2.4 behandelt des Weiteren die verschiedenen Quellkonfigurationsklassen. Abschließend wird in 3.2.5 die Filterketteninfrastruktur präsentiert.

3.2.1 Metadaten

Die Definition aller Metadaten erfolgt innerhalb der spezifischen Konfigurationsklassen, die der Abschnitt 3.2.4 behandelt. Zunächst werden die Schemata, anschließend die ergänzenden Metadatenobjekte vorgestellt.

ODS-Schemata

Es können pro Datenquelle drei verschiedene Schemata gespeichert werden. Das externe Schema beschreibt das Format der einzulesenden Daten. Die anderen beiden Schemata definieren, wie die Daten innerhalb der Datenbank des Open Data Service abgelegt und an die Nutzer ausgeliefert werden. Eines davon modelliert die Rohdaten, das andere die verbesserten Daten.

Die Schemata sind den JSON-Datentypen nachempfunden. Daher existieren einerseits die Container-Typen *MapType* und *ListType*, deren Elemente wiederum auf andere *Types* verweisen. Andererseits gibt es die primitiven Datentypen *StringType*, *NumberType*, *BoolType* und *NullType*. Die Schemastruktur ist in Abbildung 3.1 dargestellt. Ein Datenbank-Dokument ist auf der äußersten Ebene immer vom Typ *MapType*.

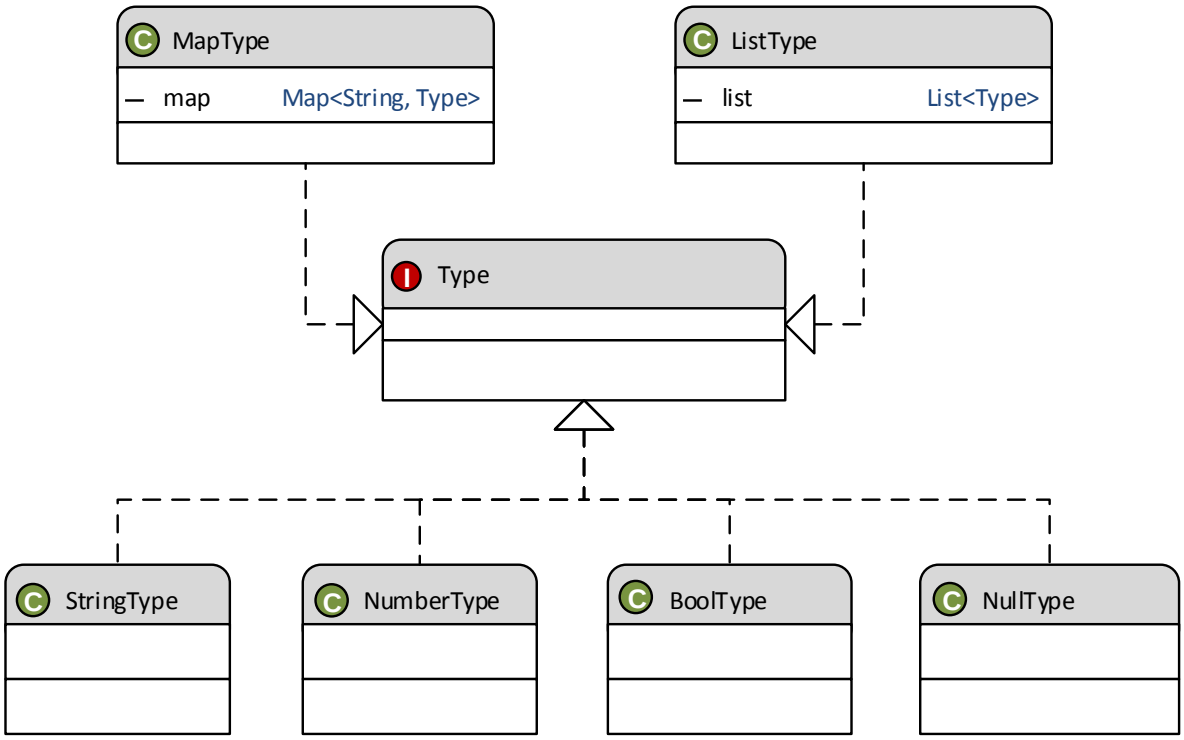


Bild 3.1: Klassendiagramm der Typhierarchie

In 2.4.2.1 wurden die Datenqualitätsfilter vorgestellt. Diese werden durch Änderungsstrukturen konfiguriert, welche Schemata darstellen, die auf die zu bearbeitenden Felder reduziert worden sind. Die Strukturen sind wie alle Schemata aus *Type*-Objekten aufgebaut und werden dynamisch in den Konfigurationsklassen definiert.

Sonstige Metadaten

Die in Abbildung 3.2 modellierte Klasse *OdsMetaData* enthält sonstige Metadaten zur Beschreibung der Quelle. Die Felder umfassen die ID, den Namen, die Beschreibung, den Autor, die Kontaktdaten des Autors, die Webseite des Anbieters, die Nutzungsbedingungen der Quellseite und den Aktualisierungszeitpunkt dieser Informationen. Bis auf die automatisch generierte Aktualisierungszeit werden diese Daten im Konstruktor übergeben. Des Weiteren existieren für die Felder passende Getter-Methoden.

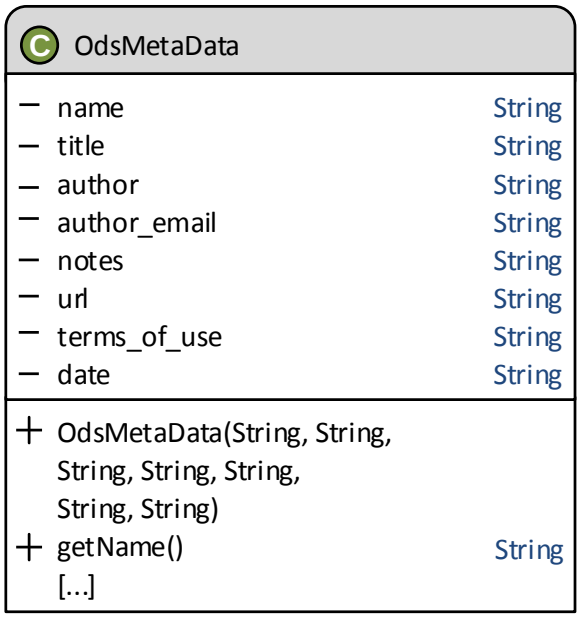


Bild 3.2: Klassendiagramm von *OdsMetaData*

Listing 3.1 zeigt das Metadaten-Dokument von *PEGELONLINE*. Genau wie die Schemata können auch die restlichen Metadaten am ODS-Server abgefragt werden.

```
{
"date": "2014-09-27␣10:23:57.004",
"name": "de-pegelonline",
"title": "pegelonline",
"author": "Wasser-␣und␣Schifffahrtsverwaltung␣des␣Bundes␣(WSV)",
"author_email": "https://www.pegelonline.wsv.de/adminmail",
"notes": "PEGELONLINE␣stellt␣kostenfrei␣tagesaktuelle␣Rohwerte␣
   verschiedener␣gewaesserkundlicher␣Parameter␣(z.B.␣Wasserstand)␣
   der␣Binnen-␣und␣Kuestenpegel␣der␣Wasserstrassen␣des␣Bundes␣bis␣
   maximal␣30␣Tage␣rueckwirkend␣zur␣Ansicht␣und␣zum␣Download␣
   bereit.",
"url": "https://www.pegelonline.wsv.de",
"terms_of_use": "http://www.pegelonline.wsv.de/gast/
   nutzungsbedingungen"
}
```

Listing 3.1: Metadaten-Dokument von PEGELONLINE

3.2.2 Datenzugriffsschicht

Die Datenzugriffsschicht des Open Data Service besteht im Wesentlichen aus vier interagierenden Klassenstrukturen. Diese werden nun nacheinander beschrieben.

CouchDbAccessor

Die erste Klasse ist der *CouchDbAccessor*, welcher die Schnittstelle *DbAccessor* umsetzt und damit die CRUD-Funktionen der *CouchDB*-Datenbank anbindet. Abbildung 3.3 zeigt die zugehörige CRC-Karte. Andere Implementierungen der Schnittstelle würden durch eine vergleichbare Karte dargestellt werden und sich nur in den Partnerklassen darunterliegender Schichten unterscheiden. Der *CouchDbAccessor* ist zum einen für das Herstellen und Trennen der Verbindung zur Datenbanken zuständig. Zum anderen behandelt er die CRUD-Operationen der *CouchDB*. Er setzt also das Erstellen, Abrufen, Aktualisieren und Löschen von Dokumenten um. Für den Abruf von Dokumenten bietet er außerdem Operationen zur Ausführung von vordefinierten Datenbankabfragen an. Die Partnerklassen des *CouchDbAccessor* umfassen einerseits verschiedene Klassen der Bibliothek *Ektorp* (siehe 2.3.1.1), die für den Zugriff auf die *CouchDB*-Datenbank nötig sind. Beim Datenimport wird der *DbAccessor* vom *DbInsertionFilter* (vergleiche 3.2.2) aufgerufen, welcher die Dokumente in die Datenbank eingefügt. Serverseitig sind die Partner schließlich die verschiedenen *Restlets*. Diese dienen dem Abfragen und Ausliefern von Daten des ODS und wurden in 2.4.1.1 genauer besprochen.

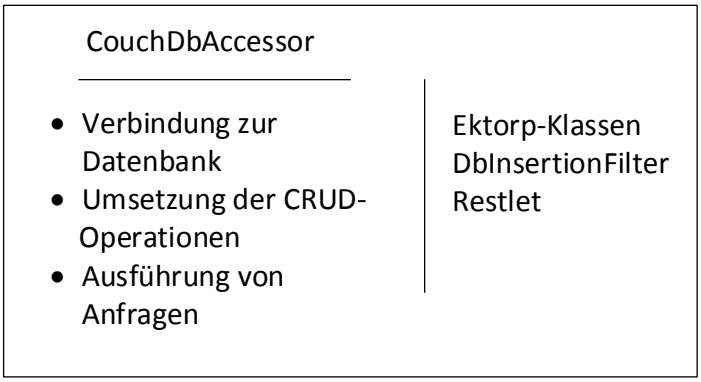


Bild 3.3: CRC-Karte der Klasse *CouchDbAccessor*

Das UML-Diagramm 3.4 beschreibt die Klasse und ihre zugehörige Schnittstelle nun im Detail. Die Methode *executeDocumentQuery* dient dazu, vordefinierte Abfragen auszuführen und eine Ergebnisliste zurückgegeben. Ihr wird der Name der Abfrage, ein Suchschlüssel (beispielsweise der Name einer Pegelmessstation, deren Wert gewünscht wird) und der Name des Dokuments, in dem die Anfrage definiert ist, übergeben. Mithilfe von *executeBulk* können außerdem mehrere Dokumente gleichzeitig in die Datenbank eingefügt oder aktualisiert werden. Dies stellt die bevorzugte Methode im Open Data

Service dar. Neben einer Menge von Dokumenten kann hier außerdem ein zu validierendes Schema (vergleiche 3.2.1) übergeben werden.

Der *CouchDbAccessor* setzt diese Schnittstelle für die Datenbank *CouchDB* um. Die Verbindung zum *CouchDB*-Server erfolgt über eine *CouchDbInstance*. Beim Öffnen einer Datenbank durch Angabe ihres Namens erhält man schließlich ein *CouchDbConnector*-Objekt, welches als Basis für den Datenbankzugriff verwendet wird. Die weiteren Methoden dienen der Erstellung einer neuen Datenbank und der Prüfung der Verbindung.

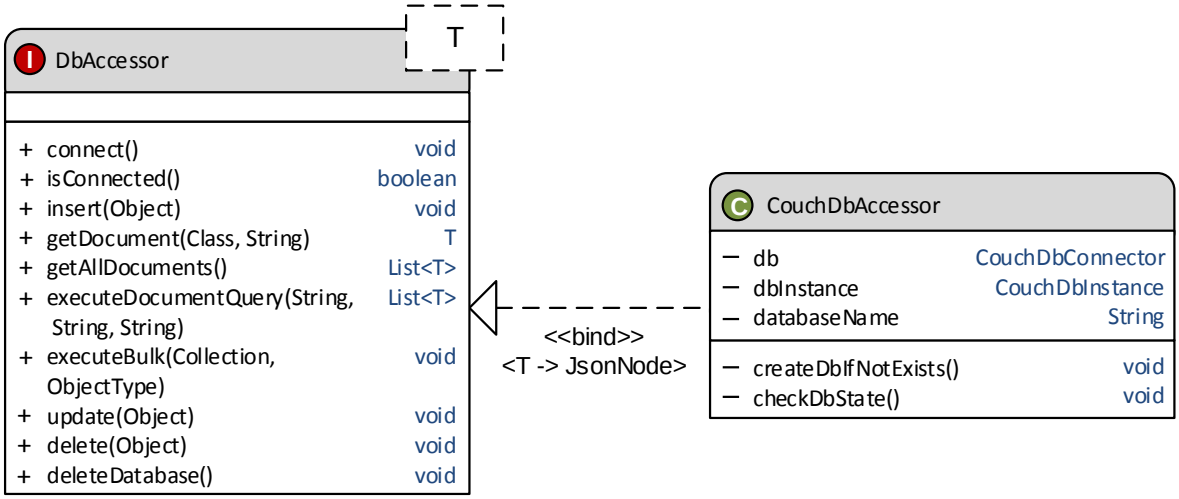


Bild 3.4: Klassendiagramm des *CouchDbAccessor*

Zusätzlich dazu existiert auch ein *MockDbAccessor*, welcher das gleiche Interface implementiert. Er wird für die Entwicklung von Modultests benötigt, da er die darunter liegende Datenbank entkoppelt und deren Verhalten stattdessen simuliert. Dadurch können diese Tests unabhängig von einer vorhandenen Datenbank auf beliebigen Rechnern durchgeführt werden.

DbFactory

Die nächste Klasse der Datenzugriffsschicht ist die *DbFactory*, dargestellt in Abbildung 3.5. Sie entkoppelt mithilfe des *Factory*-Entwurfsmusters (Gamma et al., 1994) die Wahl des konkreten *DbAccessor* von der restlichen Software. Weiterhin wird aus Konsistenzgründen der *MockDbAccessor* ebenfalls in der *DbFactory* erstellt. Die Klasse *DbFactory* wird immer dann aufgerufen, wenn Software-Komponenten Datenbankzugriff mithilfe eines *DbAccessor* benötigen.

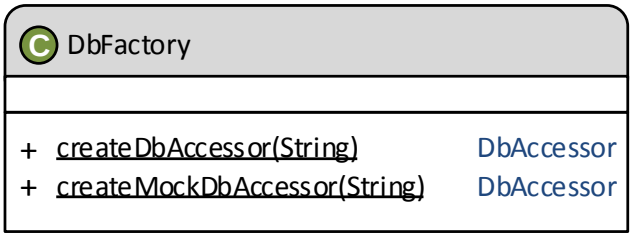


Bild 3.5: Klassendiagramm der *DbFactory*

DbInsertionFilter

Der *DbInsertionFilter* ist ein Bestandteil der in 2.3.1.2 vorgestellten und in 2.4.1.2 sowie später in 3.2.5 detaillierter beschriebenen Filterkette der Importkomponente. Die aus CRC-Karte 3.6 ersichtliche Funktionalität besteht darin, sowohl die Rohdaten als auch die verbesserten Daten in die Datenbank einzupflegen. Er nutzt dazu einen *DbAccessor*. Die Filterketten werden in einer quellspezifischen Klasse, die das *Configuration*-Interface implementiert, zusammengestellt. Der Aufruf aller *Filter* und damit auch der des *DbInsertionFilter* erfolgt innerhalb einer Instanz der Klasse *FilterChain*. Die Quellkonfigurationsobjekte sowie die Filterketteninfrastruktur werden im Anschluss noch genauer besprochen (vergleiche 3.2.4 und 3.2.5).

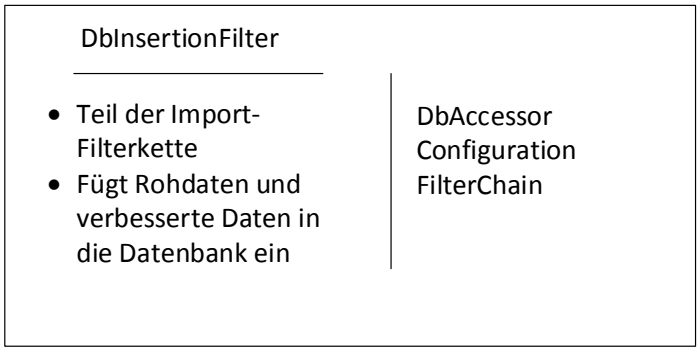


Bild 3.6: CRC-Karte der Klasse *DbInsertionFilter*

Abbildung 3.7 zeigt das zugehörige Klassendiagramm. Das referenzierte *DataSource*-Objekt ist wie das *Configuration*-Interface Teil der Quelldatenkonfiguration, die in 3.2.4 thematisiert wird. Aus diesem werden die Schemata extrahiert und ebenfalls die Datenbank eingefügt. Die Typparameter T und U der Filterschnittstelle bestehen aus den Datentypen des Inputs und des Outputs. Im Fall des *DbInsertionFilter* sind beliebige Datentypen zugelassen, da die Klasse unabhängig von einer konkreten Datenbank im-

plementiert ist. Eine Rückgabe der Objekte ist nötig, da nach dem hier stattfindenden Einfügen der Rohdaten insbesondere noch Qualitätsverbesserungen durchgeführt werden.

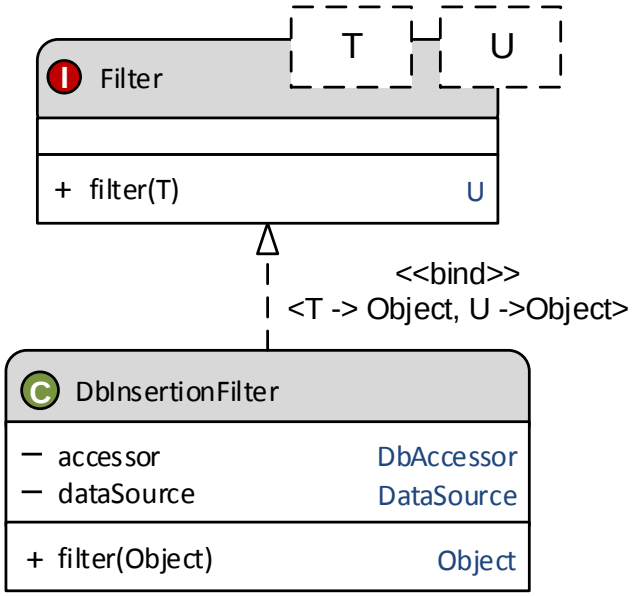


Bild 3.7: Klassendiagramm des *DbInsertionFilter*

CouchDbUtils

Die Klasse *CouchDbUtils* (Abbildung 3.8) umfasst zusätzliche Funktionalität der Datenzugriffsschicht für den Fall der Benutzung einer *CouchDB*-Datenbank. Ihre Aufgabe ist es, die vordefinierten *CouchDB*-Datenbankabfragen in speziellen Dokumenten, sogenannten Design-Dokumenten (vergleiche nächster Teilabschnitt), mithilfe des in 3.2.2 vorgestellten *CouchDbAccessor* abzuspeichern, und so zur Ausführung zur Verfügung zu stellen. Die Klasse wird bei der Initialisierung des Open Data Service durch den in 3.2.4 behandelten *ConfigurationManager* aufgerufen, da zu diesem Zeitpunkt die grundlegenden Datenbankabfragen definiert werden.

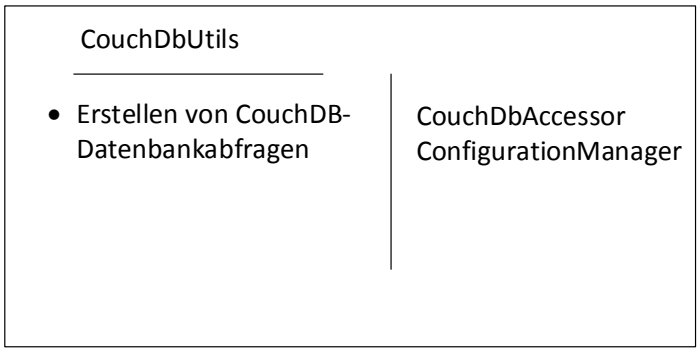


Bild 3.8: CRC-Karte der Klasse *CouchDbUtils*

CouchDB-Design-Dokumente

Listing 3.2 zeigt einen Auszug aus dem PEGELONLINE-Design-Dokument, welches der Definition quellspezifischer Datenbankabfragen dient. Die Abfragen werden hierfür im Feld *views* abgelegt. Sie enthalten als Schlüssel ihren Namen und als Werte die in *JavaScript* implementierten Funktionen. *CouchDB* setzt dabei das *MapReduce*-Prinzip (Dean and Ghemawat, 2008) ein, wobei im Rahmen des ODS bislang nur *Map*-Funktionen benötigt werden. Beim Aufruf einer im Design-Dokument definierten Operation führt *CouchDB* diese für jedes einzelne Datenbankdokument aus.

Die Methode *getAllStations* prüft für jedes Dokument, ob es sich um eine Station handelt, und fügt es im positiven Fall mithilfe der eingebauten *emit*-Funktion zur Ergebnismenge hinzu. *getSingleStation* dient der Abfrage von Dokumenten eines speziellen Stationsnamens. Der übergebene Schlüssel wird für alle Stationsobjekte mit dem Inhalt des Felds *longname* (Name der Station) verglichen. Bei Übereinstimmung wird das Dokument zur Ergebnismenge hinzugefügt. Mit *getMetadata* und *getSchema* können außerdem Metadaten- respektive Schemadokumente abgefragt werden.

```
{
      [...]
   "views": {
        "getAllStations": {
            "map": "function(doc) { if(doc.dataType == 'Station') emit
               (null, doc) }"
        },
        "getSingleStation": {
            "map": "function(doc) { if(doc.dataType == 'Station') emit(
               doc.longname, doc)}"
        },
        "getMetadata": {
            "map": "function(doc) { if(doc.title == 'pegelonline') emit
               (null, doc)}"
        },
        "getSchema": {
            "map": "function(doc) { if(doc.name == 'de-pegelonline-
               station') emit (null, doc) }"
        }
   },
      [...]
}
```

Listing 3.2: Auszug aus dem Design-Dokument für PEGELONLINE-Daten

3.2.3 Übersicht über die Server-Schnittstelle des Open Data Service

Die REST-Schnittstelle des ODS bietet eine Vielzahl von Funktionen zur Abfrage von Datensätzen an. Ein Auszug der wichtigsten Operationen ist in Listing 3.3 dargestellt. Die Schnittstellenbeschreibung ist das Ergebnis der Anfrage nach der Ressource */api*. Mittels */ods* kann man ODS-übergreifend auf der obersten Dokumentebene nach Schlüssel-Wert-Paaren suchen. */ods?dataType=Osm* listet beispielsweise alle gespeicherten OSM-Dokumente auf. */ods/${id}* beschreibt die Adressen für den ID-Zugriff, wobei id eine Variable darstellt, die in der Anfrage durch die gewünschte Dokument-ID ersetzt wird. Diese Funktionalität wird zwar bereits durch die Möglichkeit der Angabe von Schlüssel-Wert-Paaren erreicht, die alternative Methode stellt jedoch eine elegantere Umsetzung des REST-Paradigmas dar. Ein Aufruf der Adresse */ods/schema* gibt des Weiteren alle vorhandenen Schemata aus.

Darüber hinaus existieren auch quellspezifische API-Zweige. So können beispielsweise für *PEGELONLINE* und *Pegelportal Mecklenburg-Vorpommern* alle Stationen, einzelne Stationen, das Stationsschema und die Metadaten abgefragt werden. Auch eine flache Übersicht über die Namen aller vorhandenen Stationen ist verfügbar.

Falls verbesserte Dokumente existieren, werden stets diese zurückgegeben. Für einen expliziten Abruf der Rohdaten muss die Option *?dataQualityStatus=raw* gewählt werden.

```
{
    [
    "/api",
    "/ods",
    "/ods/${id}",
    "/ods/schema",
    "/ods/de/pegelonline/stations",
    "/ods/de/pegelonline/stationsFlat",
    "/ods/de/pegelonline/stations/$schema",
    "/ods/de/pegelonline/stations/{station}",
    "/ods/de/pegelonline/metadata",
    "/ods/de/pegelportal-mv/stations",
    "/ods/de/pegelportal-mv/stationsFlat",
    "/ods/de/pegelportal-mv/stations/$schema",
    "/ods/de/pegelportal-mv/stations/{station}",
    "/ods/de/pegelportal-mv/metadata",
    "/ods/de/osm",
    "/ods/de/osm/$schema",
    "/ods/de/osm/data/{osm_id}",
    "/ods/de/osm/relations/{osm_id}",
    "/ods/de/osm/ways/{osm_id}",
    "/ods/de/osm/nodes/{osm_id}",
    "/ods/de/osm/metadata",
    [...]
    ]
}
```

Listing 3.3: Auszug der Server-Schnittstelle

3.2.4 Beschreibung und Konfiguration der Quelldaten für die Importkomponente

Der erste Schritt der in 3.2.5 beschriebenen Filterkette ist das Herunterladen von Quelldaten. Dazu werden verschiedene Informationen benötigt, die derzeit in Klassen definiert sind, die das *Configuration*-Interface implementieren. Die Schnittstelle respektive deren Umsetzungen werden mithilfe der CRC-Karte in Abbildung 3.9 und des UML-Diagramms 3.10 vorgestellt. Ihre Aufgabe ist es, die Filterkette sowie ein *DataSource*-Objekt zu konfigurieren. Das Konzept der Filterkette wurde dabei in 2.3.1.2 vorgestellt, ihr konkreter Entwurf wird im Anschluss in 3.2.5 besprochen. Die Kette wird in der Methode *getFilterChain* individuell zusammengestellt. In *getDataSource* wird nun das ebenfalls noch beschriebene *DataSource*-Objekt definiert, welches die Internet-Adresse sowie Schemata

und weitere Metadaten der Quelle (vergleiche 3.2.1) definiert. Erstellt und zusammengeführt werden die *Configurations* im *ConfigurationManager*.

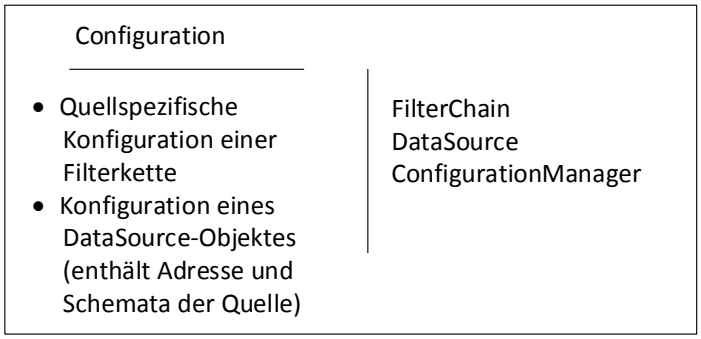


Bild 3.9: CRC-Karte der Schnittstelle *Configuration*

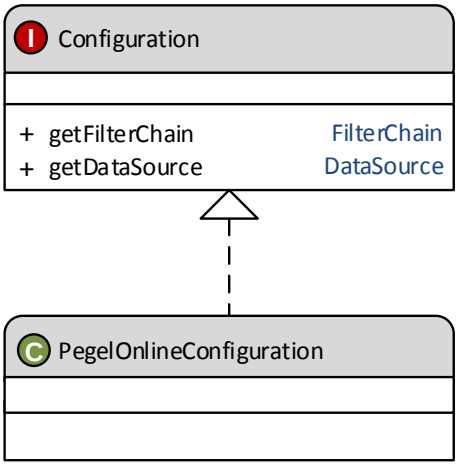


Bild 3.10: Klassendiagramm der *Configuration*-Schnittstelle

Abbildungen 3.11 und 3.12 zeigen CRC-Karte und Klassendiagramm von *DataSource*. Ein *DataSource*-Objekt enthält eine, im ODS eindeutige, ID, die Adresse der Quelle, das Quellschema, die Schemata der in der Datenbank abgelegten Dokumente in ursprünglicher und verbesserter Form, sonstige Metadaten (vergleiche 3.2.1) sowie eine Liste von quellspezifischen Datenbankabfragen (3.2.2). Die Definition der *DataSources* erfolgt in den zuletzt beschriebenen Konfigurationsobjekten. Verwendet werden sie zur Ermittlung der Quelladressen im *Grabber* (2.4.1.2) und zum Einfügen der Metadaten in die Datenbank innerhalb des *DbInsertionFilter* (3.2.2). Neben dem Konstruktor existieren auch die passenden Getter-Methoden.

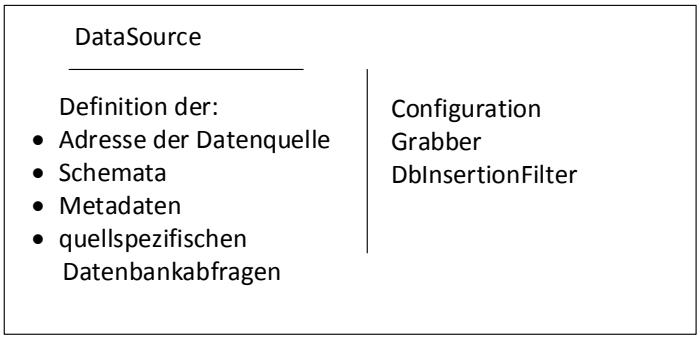


Bild 3.11: CRC-Karte von *DataSource*

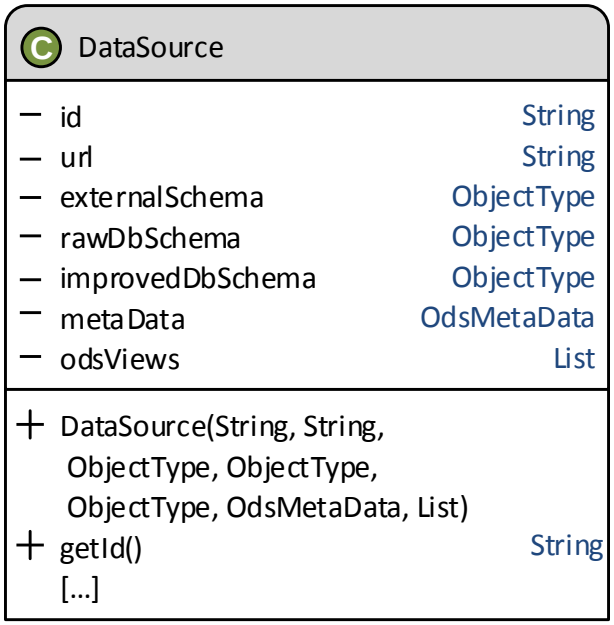


Bild 3.12: Klassendiagramm der *DataSource*

Die verschiedenen Konfigurationen werden von der Klasse *ConfigurationManager* verwaltet, die durch in den Abbildungen 3.13 und 3.14 modelliert wird. Die Aufgabe der Klasse ist es, die einzelnen Quellkonfigurationen (die Liste *configurations*), inklusive der darin enthaltenen Filterketten, in der Methode *configureAll* zu erstellen und dann jeweils beim *FilterChainManager* (vergleiche 3.2.5) zu registrieren.

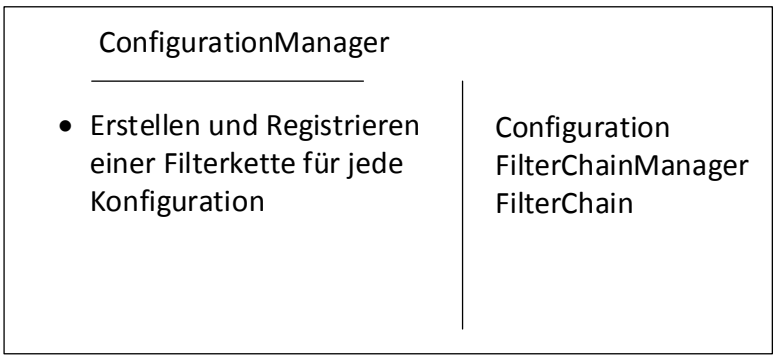


Bild 3.13: CRC-Karte der Klasse *ConfigurationManager*

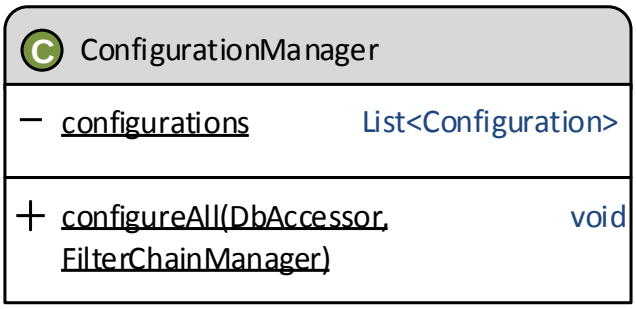


Bild 3.14: Klassendiagramm des *ConfigurationManager*

3.2.5 Die Filterketteninfrastruktur der Importkomponente

Die Infrastruktur der Filterkette besteht im Wesentlichen aus dem Zusammenspiel von drei Komponenten, dem *FilterChainManager* (siehe Bild 3.15), der *FilterChain* (3.16) und dem *Filter*-Interface. Der *FilterChainManager* enthält für jede Quelle eine *FilterChain*, welche ihrerseits aus den einzelnen in 2.3.1.2 vorgestellten *Filtern* besteht. Die Menge der quellspezifischen Filterketten wird im Rahmen der Arbeit als äußere Filterkette bezeichnet.

Der *FilterChainManager* interagiert zum einen mit dem in 3.2.4 vorgestellten *ConfigurationManager*, welcher für jede Quelle deren *FilterChain* registriert. Zum anderen initiiert die Hauptklasse der Importkomponente, *ImportMain* (siehe 2.4.1.2), den gesamten Datenimportvorgang, indem die Methode *startFilterChains* aufgerufen wird. Innerhalb des *FilterChainManager* werden dann die quellspezifischen *FilterChains* gestartet. Die ebenfalls in einer *Configuration* definierte *FilterChain* startet nun wiederum nacheinander ihre zugeordneten *Filter*. Diese führen schließlich ihre jeweiligen Änderungsoperationen auf den Daten durch.

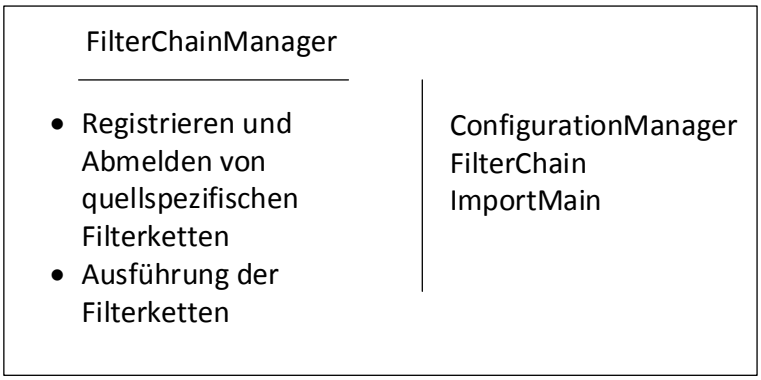


Bild 3.15: CRC-Karte der Klasse *FilterChainManager*

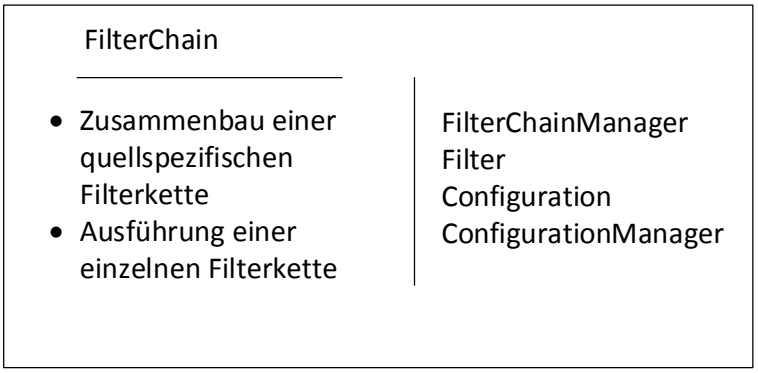


Bild 3.16: CRC-Karte der Klasse *FilterChain*

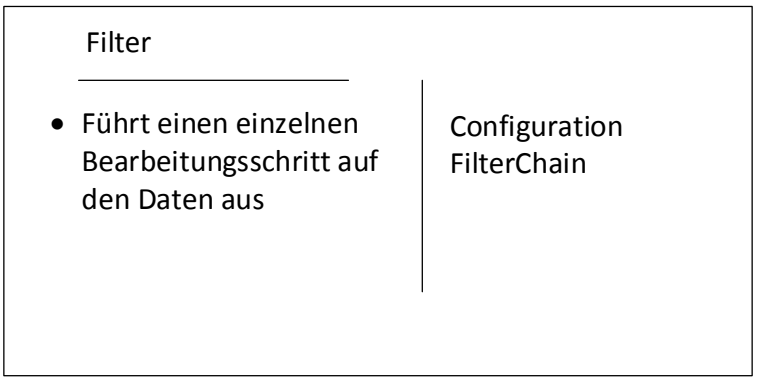


Bild 3.17: CRC-Karte der Schnittstelle *Filter*

Die Klassendiagramme von *FilterChainManager*, *FilterChain* und *Filter* sind in den Abbildungen 3.18, 3.19 sowie 3.20 zu finden. Der *FilterChainManager* enthält die Methoden zur Verwaltung und Initiierung der einzelnen Filterketten. Die *FilterChain* baut mithilfe der Methode *setNextFilter* die Filterkette auf, wobei jeweils ein Element aus der Filterkette nach dem Zwiebelprinzip hinzugefügt und in der *filter*-Methode später wieder ausgepackt wird. Die *filter*-Methode von *FilterChain* erstellt eine Kopie der zu behandelnden Daten und übergibt diese dann an die zu unterscheidende, ebenfalls *filter* genannte, Methode der *Filter*-Schnittstelle. Das *Filter*-Interface besitzt zwei Typparameter, welche den Eingabe- respektive Ausgabetyp des jeweiligen Filterungsschritts repräsentieren.

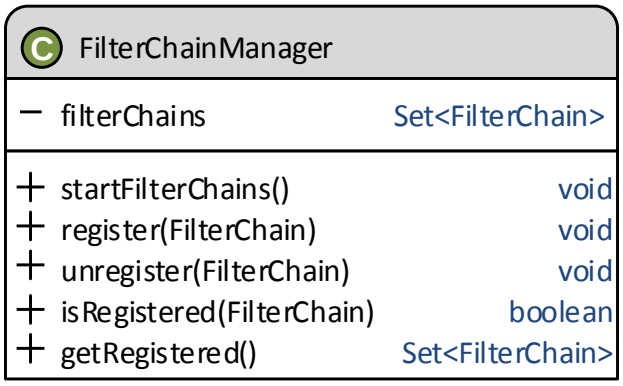


Bild 3.18: Klassendiagramm des *FilterChainManager*

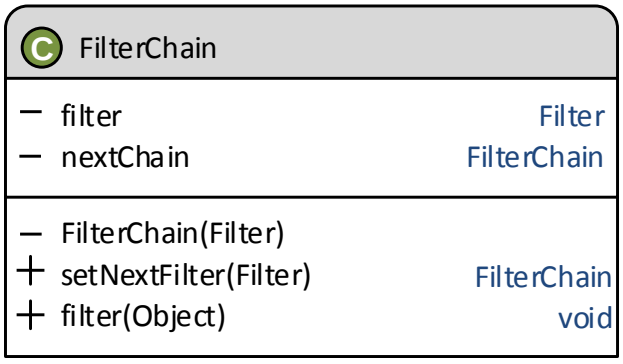


Bild 3.19: Klassendiagramm der *FilterChain*

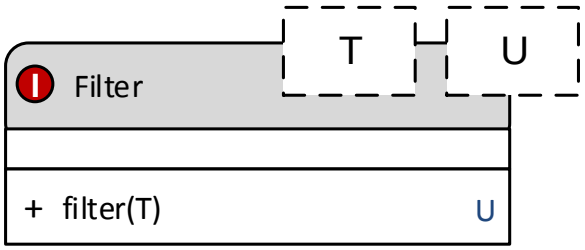


Bild 3.20: Klassendiagramm der *Filter*-Schnittstelle

Literaturverzeichnis

Aamodt, A. and Nygård, M. (1995). Different roles and mutual dependencies of data, information, and knowledge — an AI perspective on their integration. *Data and Knowledge Engineering*, 16(3):191–222.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1.

Bäumer, D., Riehle, D., Siberski, W., Lilienthal, C., Megert, D., Sylla, K.-H., and Züllighoven, H. (1998). Values in Object Systems.

Beck, K. and Cunningham, W. (1989). A laboratory for teaching object oriented thinking. In *ACM Sigplan Notices*, volume 24, pages 1–6. ACM.

Bellinger, G., Castro, D., and Mills, A. (2004). Data, information, knowledge, and wisdom.

Buschmann, F., Henney, K., and Schmidt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John Wiley & Sons.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27.

Cunningham, W. (1995). The CHECKS pattern language of information integrity. In *Pattern languages of program design*, pages 145–155. ACM Press/Addison-Wesley Publishing Co.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM.

Fielding, R. and Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gantz, J. and Reinsel, D. (2011). Extracting value from chaos. *IDC iview*, pages 1–12.

Garlan, D. and Shaw, M. (1994). An introduction to software architecture.

Go, A., Bhayani, R., and Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, pages 1–12.

Haklay, M. (2010). How good is volunteered geographical information? A comparative study of OpenStreetMap and Ordnance Survey datasets. *Environment and planning. B, Planning and design*, 37(4):682.

Haklay, M. and Weber, P. (2008). OpenStreetMap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18.

IEEE 802.3 Ethernet Working Group (2012). IEEE 802.3 TM Industry Connections Ethernet Bandwidth Assessment.

ISO 9000 (2005). ISO 9000:2005 Quality management systems – Fundamentals and vocabulary. `http://www.iso.org/iso/catalogue_detail?csnumber=42180`.

ISO/IEC 23270 (2006). ISO/IEC 23270:2006(E) Information technology — Programming languages — C#. `http://www.iso.org/iso/catalogue_detail?csnumber=42926`.

Ko, M. N., Cheek, G. P., Shehab, M., and Sandhu, R. (2010). Social-networks connect services. *Computer*, 43(8):37–43.

Lin, C. and Atkin, D. (2007). *Communication Technology and Social Change: Theory and Implications*. Lawrence Erlbaum Associates.

Makice, K. (2009). *Twitter API: Up and running: Learn how to build applications with the Twitter API*. O'Reilly Media, Inc.

Merz, B., Elmer, F., Kunz, M., Mühr, B., Schröter, K., and Uhlemann-Elmer, S. (2014). The extreme flood in June 2013 in Germany. *La Houille Blanche*, pages 5–10.

Moore, G. E. et al. (1965). Cramming more components onto integrated circuits.

O'Reilly, T. (2007). What is Web 2.0: Design patterns and business models for the next generation of software. *Communications and Strategies*, 65(1):17–37.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. "big" web services: Making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly Media, Inc.

Riehle, D. (2006). Value object. In *Proceedings of the 2006 conference on Pattern languages of programs*, page 30. ACM.

Riehle, D. (2011). Lessons learned from using design patterns in industry projects. In *Transactions on pattern languages of programming II*, pages 1–15. Springer.

Roberts, L. G. (2000). Beyond moore's law: Internet growth trends. *Computer*, 33(1):117–119.

Saake, G., Schmitt, I., and Türker, C. (1997). *Objektdatenbanken: Konzepte, Sprachen, Architekturen*. Internat. Thomson Publ.

Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11.

Te Linde, A., Bubeck, P., Dekkers, J., De Moel, H., and Aerts, J. (2011). Future flood risk estimates along the river Rhine. *Natural Hazards and Earth System Sciences*, 11:459–473.

Thompson, A. and Taylor, B. N. (2008). Guide for the Use of the International System of Units. *NIST Special Publication*, 811.

Wang, R. Y. (1998). A product perspective on total data quality management. *Communications of the ACM*, 41(2):58–65.

Wang, R. Y. and Strong, D. M. (1996). Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, pages 5–33.

Webster, F. (2014). *Theories of the information society*. Routledge.