# Serialisierung von Datenobjekten in JSON zur Übertragung von Objekten aus Energieanwendungen

## Projektarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Sebastian Rieger**

Abgabedatum 15. September 2014

| | |
|---|---|
| Bearbeitungszeitraum | 13 Wochen |
| Matrikelnummer | 7406886 |
| Kurs | TINF12B1 |
| Ausbildungsfirma | Karlsruher Institut für Technologie (KIT) Karlsruhe |
| Betreuer der Ausbildungsfirma | Dr.-Ing. Karl-Uwe Stucky |

## Erklärung

Gemäß§16 (3) der „Studien- und Prüfungsordnung für den Studienbereich Technik“ vom 1.11.2007.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

_____ _____

Ort Datum Unterschrift

# Inhaltsverzeichnis

# 1 Einleitung

Die folgende Arbeit befasst sich mit der Serialisierung und Deserialisierung von Datenobjekten in *JavaScript Object Notation* (JSON)-Format zur Übertragung von Objektdaten, die sogenannten Anwendungsdaten, welche in Programmen generiert werden.

## 1.1 Generisches Managementsystem für Energiedaten

*Generic Data Services* (GDS) ist das generisches Managementsystem für Energiedaten. Es verwendet Objekte, um Anwendungsdaten zu verwalten, die einem *Objektorientierten Programmiermodell* (OPM) genügen und durch *Strukturelle Metadaten* (SMD) beschrieben werden können.

Alle Programmbestandteile des GDS sind durch das objektorientierte Datenmodell programmiersprachenunabhängig und können durch strukturelle Metadaten beschrieben werden.

Das GDS wird zur Zeit am *Institut für Angewandte Informatik* (IAI) des *Karlsruher Instituts für Technologie* (KIT) entwickelt und ist ein System generischer Datenservices, welches bei Fertigstellung vollautomatisch Energiedaten managen soll.

Durch den generischen Charakter kann es aber auch Daten aus anderen Bereichen verwalten, wenn diese dem OPM-Modell genügen.

## 1.2 Objektorientiertes Datenmodell

Im OPM werden Richtlinien für die Entwicklung von allen objektorientierten Softwarebausteinen festgelegt. Das gesamte Projekt ist bisher OPM-konform gehalten und somit soll auch die Schnittstelle der Serialisierung OPM-konform gestaltet werden.

OPM ist im weiteren Sinn eine Abstraktionsschicht, mit dessen Hilfe ein Programm programmiersprachenunabhängig beschrieben werden kann.

Um diese Unabhängigkeit zu erreichen, umfasst das Modell viele Regeln, wie Programmcode aussehen soll und welchen Anforerungen er genügen muss.

Im Folgenden sind die wichtigsten OPM-Regeln dargstellt:

- Basisklasse `OPMObject` von der alle Klassen erben
- Es gibt keine Konstanten
- Attribute sind grundsätzlich `private`, also von außerhalb der Klasse nicht änderbar, und werden gegebenenfalls durch Getter- und Setter-Methoden aufgerufen (`private` also nicht im Sinn von Java)
- Programme bestehen nur aus Objekten, der Aufruf erfolgt ausschließlich über Methodenaufrufe
- Es können Standarddatentypen der jeweiligen Programmiersprache verwendet werden
- Methodenbezeichner werden im „Camel Case“ formuliert
- Attributbezeichner werden im „Lower Camel Case“ formuliert
- In der Dokumentation eines Attributs wird immer der erlaubte Wertebereich spezifiziert
- Die Dokumentation des Programmcodes muss den Spezifikationen der verwendeten Sprache entsprechen, damit diese auch als Dokumentation vom Compiler erkannt wird

- Jede Klasse wird mit einem Status `Valid`, `Experimental` oder `Deprecated` in ihrer Dokumentation beschrieben, um den Entwicklungsstand sofort erkennen zu können

Alle Klassen, die vom GDS verwaltet werden sollen, müssen diesen Grundsätzen genügen, um verarbeitet werden zu können.

Des Weiteren gibt es, wie eben schon kurz dargestellt, im OPM eine Oberklasse `OPMObject`, von der alle Klassen erben sollen. Dieses Vorgehen ist zum Beispiel auch in Java umgesetzt, wo jede Klasse von `Object` erbt.

Vorteil dieses Konzepts ist, dass Methoden, Konstruktoren und Attribute, welche jede Klasse haben soll, nur einmal in der Oberklasse aufgeführt werden müssen und dann vererbt werden.

## 1.3 Strukturelle Metadaten

Strukturelle Metadaten sind laut der OPM-Definition spezielle Metadaten, die den Aufbau einer Programmklasse enthalten. Die Informationen der SMD sind programmiersprachenunabhängig und stehen den Anwendungsprogrammen zur Verfügung. Die strukturellen Metadaten sind für jede Klasse vorhanden und enthalten unter anderem Attributnamen, Attributtyp und den dazugehörigen Qualifier, außerdem Methodennamen, Methodenattribute, den Rückgabewert und den Quallifier. Siehe Kapitel 6.

Die Metadaten werden in einer SQL-Datenbank gespeichert, aus der sie, wenn benötigt, geladen werden können. Diese Arbeit übernimmt ein Assistent, welcher im Kapitel 6 noch genauer beschrieben wird. [Zil14]

# 2 Aufgabenstellung

Im GDS-System werden Anwendungsdaten mit Hilfe von Klassen-Objekten verwaltet, welche den Regeln von OPM folgen und mittels der SMD genau beschrieben werden können.

Für den Transfer von solchen Anwendungsdaten sind Schnittstellen für die automatische Serialisierung der solcher Daten vorgesehen, welche im Rahmen dieser und einer weiteren Arbeit entwickelt werden sollen. [Wal14]

Ziel der Arbeit ist es, Spannungszeitreihen und OPM-Objekte mittels der Metadaten in ein JSON-Format zu serialisieren und diese zu übertragen. Allgemein muss jede im Projekt vorkommende Klasse serialisiert werden können. Die Empfängerseite muss letztendlich in der Lage sein, aus den übertragenen JSON-Daten wieder Objekte zu erstellen, welche im Programm weiter verarbeitet werden können. Da es sich im Projekt bei allen Klassen um OPM-Objekte handelt, muss es prinzipiell möglich sein, jedes Klassen-Objekt zu serialisieren.

Es soll analysiert werden, welche Möglichkeiten bestehen, um Java-Klassen-Objekte zu serialisieren, beziehungsweise zu deserialisieren. Hierfür soll ein Vergleich verschiedener Techniken erfolgen, welche zu bewerten sind.

Die beste Möglichkeit soll im Anschluss der Überprüfung implementiert und genauer untersucht werden. Hierbei ist ein besonderes Augenmerk auf die Geschwindigkeit des Serialisierungprozesses zu legen. Aber auch grundlegende Aspeke wie Einfachheit der Implementierung, Zusammenspiel mit anderen Serialisierern und die Anwendbarkeit an realitätsnahen Daten soll überprüft werden.

Es soll des Weiteren geprüft werden, ob alle „Standard-Datentypen“ serialisiert werden können und wie der Umgang mit Klassen-Attributen, Rekursion und Attributen, die nicht serialisiert werden sollen, beschaffen ist.

Abschließend soll ein Vergleich zeigen, welche Serialisierungsart (JSON oder XML), für das Projekt, besser geeignet ist. Hierfür wird die Arbeit „Serialisierung von Datenobjekten in XML zur Übertragung von Objekten aus Energieanwendungen“ von Herrn Achim Walz herangezogen. [Wal14]

# 3 JavaScript Object Notation

JSON ist ein textbasiertes Format zum Datenaustausch, wobei jedes gültige JSON-Dokument auch ein gültiges JavaScript ist. Jedoch ist JSON an sich unabhängig von der Programmiersprache überall einsetzbar.

Es wurde als Ersatz für XML geschaffen und wird hauptsächlich in Bereichen eingesetzt wo Ressourcen wie Speicherplatz, Prozessorleistung und Netzwerkverbindung stark limitiert sind. Der Aufbau von JSON erinnert stark an die Struktur eines Arrays. Ein Beispiel für ein JSON-Objekt ist im Kapitel 3.1 zu finden. [Wik14a]

## 3.1 Der Aufbau von JSON

Ein gültiges JSON-Dokument ist im Beispiel auf der nächsten Seite zu finden. In der ersten und letzten Zeile sind geschweifte Klammern zu finden, da jedes JSON-Dokument ein Objekt, im Sinne von JSON-Objekten, ist und Objekte in JSON von geschweiften Klammern umschlossen werden müssen. [Sri13]

Die geschweiften Klammern zeigen im JSON-Format somit den den Beginn, beziehungsweise das Ende eines Objektes an. Hierbei ist zu beachten, dass Objekte auch verschachtelt auftreten können.

Um das JSON-Beispiel zu verdeutlichen, ist hier eine Java-Klasse dargestellt, die in dasJSON-Format umgewandelt wird.

Ein Beipiel für ein zu serialisierendes Java-Klassen-Objekt, ohne Annotationen für einen Serialisierer, könnte wie folgt aussehen. Worum es sich bei Annotationen genau handelt, wird im Kapitel 5 beschrieben. In der Klasse sind lediglich die zu serialisierenden Attribute dargestellt, jedoch keine Methoden, da sie für die Serialisierung nicht von Bedeutung sind.

```
1 public class JsonObjects {
2     public String stringValue = "Variable123";
3     public boolean boolValue = true;
4     public String nullValue = null;
5     public int zahlValue = 1234567;
6     public double fliesskommaValue = 1230000.0;
7     public String arrayValue[] = new String[]{"Beispiel", "fuer", "Array"};
8     public OtherObject object = new OtherObject();
9 }
```

Diese Klasse sieht nun im JSON-Format wie folgt beschrieben und gezeigt aus.

JSON ist nach dem Schlüssel/Wert-Prinzip aufgebaut, was bedeutet, dass jedem Schlüssel genau ein Wert zugeordnet werden kann. Im Beispiel sind alle in JSON möglichen Formattypen aufgezeigt.

Das Beispiel der Java-Klasse als serialisiertes JSON-Objekt würde dann wie folgt aussehen können. Je nachdem, mit welcher Bibliothek serialisiert wird, beziehungsweise welche Annotationen an den Quellcode noch zusätzlich angebracht sind, kann das JSON-Objekt auch anders aussehen.

```
1 {
2     "stringValue": "Variable123",
3     "boolValue": true,
4     "nullValue": null,
5     "zahlValue": 1234567,
6     "fliesskommavalue": 1.23e+6,
7     "arrayValue": ["Beispiel", "fuer","Array"],
8     "objekt": {
9         "zahl_2Value": "1234",
10    }
11 }
```

Zeile 2 enthält einen String, eine Zeichenkette, in dem jedes Zeichen erlaubt ist. Ein String unter JSON hat genau das selbe Aussehen und den selben Zeichensatz wie unter Java, jedoch keine Funktionalität, da JSON ein reines Datenformat ist.

Der boolsche Wert wird genau wie ein Nullwert ohne Anführungszeichen geschrieben, wie in den Zeilen 3 und 4 gezeigt. Ein boolscher Wert kann, genau wie in Java, zwei Zustände haben, nämlich `true` oder `false`.

Zahlen können ganzzahlig, Fließkommazahlen oder Exponentialzahlen sein. Die Schreibweise dafür ist in den Zeilen 5 und 6 zu finden.

Ein Array kann mehrere Werte enthalten und wird deshalb von eckigen Klammern umschlossen. Die eigentlichen Werte im Array müssen jedoch vom selben Typ sein, wie im Beipiel ein String-Array. Grundsätzlich ist aber jeder Datentyp als Array möglich, so auch ein Array von bestimmten Objekten.

Objekte können wiederum Objekte enthalten, wie es in den Zeilen 8 bis 10 dargestellt ist. Das innere Objekt wird wieder von geschweiften Klammern umschlossen, da dies der JSON-Standard ist.

Somit können sechs Datentypen in JSON unterschieden werden: Strings, Zahlen, Booleans, Arrays, Objekte und Nullwerte. Zu beachten ist, dass Booleans, Nullwerte und Zahlen ohne Anführungszeichen geschrieben werden.

## 3.2 JSON in Verbindung mit Programmiersprachen

Viele Programmiersprachen wie PHP, Python, C#, C++ und Java unterstützen JSON sehr gut und sogar nativ. Dies bedeutet, dass für eine grundlegende Verwendung von JSON keine zusätzlichen Bibliotheken benötigt werden. Somit entfällt das Einbinden fremder Bibliotheken, die eine Verarbeitung von JSON erst ermöglichen würden.[Sri13]

Wie im Kapitel 3.1 dargestellt, gibt es viele Gemeinsamkeiten zwischen JSON und modernen Programmiersprachen. Dies vereinfacht eine Verwendung zusätzlich, da Datentypen wie String, Integer, Boolean und so weiter direkt und ohne eine zusätzliche Umwandlung gelesen werden können.

Eine Verwendung von JSON ohne einen speziellen Anwendungsfall, der wirklich JSON-Objekte benötigt, wie das Verwenden von Jackson oder MongoDB, welche beide auf JSON aufbauen, ist wenig sinnvoll. Eine Kapselung von Information in JSON, ist somit nur sinnvoll, wenn auch bestimmte Programmteile dafür ausgelegt sind mit ihnen zu arbeiten.

Eine bloße Kapselung ist somit zwar möglich, aber nicht immer Sinnvoll, da hier das Umwandeln in JSON-Objekte und zurück zusätzliche Zeit in Anspruch nehmen würde. Diese zusätzlich benötigte Zeit ist nur dann Sinnvoll eingesetzt, wenn wie bei der Serialisierung, JSON-Objekte benötigt werden.

### 3.2.1 JSON und JavaScript

JSON wird unter JavaScript als ganz normale Variable geführt und kann auch als solche ausgelesen werden. Dies geschieht beispielhaft über das JavaScript-Kommando `alert(JSONVariablenName.Zahl)`. Der Aufruf liefert den Wert 1234567 aus dem Beispiel in Kapitel 3.1, unter der Bedingung, dass das JSON-Objekt als Variable mit dem Namen `JSONVariablenName` im JavaScript deklariert wurde, zurück.

Im Beispiel ist gut zu sehen, dass unter JavaScript kein weiterer Methodenaufruf benötigt wird, um direkt auf die Variable im JSON-Objekt zuzugreifen. Das Komando `alert()` ist lediglich für die Bildschimausgabe unter JavaScript zuständig.

# 4 Objektserialisierung nach JSON

Wie in der Aufgabenstellung, im Kapitel 2, vorgegeben, sollen hier nun die Möglichkeiten einer Serialisierung von Java-Objekten, welche OPM-konform sind, in JSON untersucht werden.

Da im Projekt alle Klassen OPM-konform sind, reicht diese Forderung aus, um alle vorkommenden Klassen serialisieren und deserialisieren zu können.

## 4.1 Was ist Serialisierung

Serialisierung ist die Abbildung von Daten auf eine geeignete Darstellungsform und wird oft bei verteilten Softwarelösungen, wie im Falle von GDS, verwendet. Der erzeugte Datenstrom kann dann entweder über ein Netzwerk übertragen oder lokal gespeichert werden. Somit liegt das Objekt doppelt vor, zum einen als reales Objekt eines Programms und als serialisiertes Objekt. Eine Änderung des Objekts im Programm hat keine Auswirkung auf das serialisierte Objekt. [Wik14b]

Im Rahmen dieser Arbeit heißt das, OPM-konforme, strukturierte Java-Objekte in einen JSON-Datenstrom zu wandeln. Dieser Datenstrom soll danach über eine Netzwerkverbindung übertragen und an anderer Stelle weiter verarbeitet werden können.

## 4.2 Möglichkeiten der JSON-Serialisierung in Java

Grundsätzlich gibt es verschiedene Möglichkeiten, eine JSON-Serialisierung in Java durchzuführen. Es besteht zum einen die Möglichkeit, die nötige Logik und Klassen für eine JSON-Serialisierung selber zu implementieren, zum anderen auf schon existierende Bibliotheken aufzubauen.

### 4.2.1 Eigener Ansatz

Eine Möglichkeit, einen funktionsfähigen Serialisierer zu erhalten, ist diesen selbst zu schreiben. Hierfür müsste eine Lesefunktion für JSON-Objekte implementiert werden, was auch als Scanner bezeichnet wird.

Dieser Scanner muss in der Lage sein, einen JSON-Datenstrom zu lesen und ihn in die einzelnen Bestandteile aufspalten. Dies geschieht üblicherweise mit Hilfe eines Baums, der beim Lesen erzeugt wird. Dieser Baum kann beim Umwandeln, also der eigentlichen Serialisierung, dann auf verschiedene Arten geprüft und durchlaufen werden.

Die Verwendung eines Baums ist wichtig, da durch ihn einfacher auf Verschachtelungen und innere Strukturen, im eingelesenen Objekt, geachtet werden kann als bei der linearen Verarbeitung. [Jah10]

Eine weitere Funktion, die erfüllt werden muss, ist die eines Parsers. Dieser muss die einzelnen vom Scanner erkannten Bestandteile in Java-Klassen-Objekte umwandeln. Dies kann er zum Beispiel tun, indem er den, vom Scanner erzeugten, Baum komplett durchläuft. Hierbei muss natürlich auf zuvor, vom Scanner, gelesene und im Baum abgelegte Strukturen geachtet werden.

Natürlich muss auch die umgekehrte Richtung, also Java-Klassen-Objekte in JSON-Objekte unterstützt werden. Hierbei bleiben die Funktionen vom Scanner und Parser gleich, lediglich die Transformationsrichtung ändert sich.

Bei der Implementierung muss des Weiteren zum Beispiel auf Rekursion und nicht valide JSON-Objekte geachtet werden. So muss bestimmt werden, wie tief eine Rekursion, also eine sich selbst

aufrufenden Funktion im Programm, bearbeitet wird. Aber auch nicht valide JSON-Dokumente müssen abgefangen werden und ein sinnvoller Fehler muss ausgegeben werden.

Es gibt mehrere Möglichkeiten, eine Rekursion zu lösen. Eine wäre, wie schon erwähnt, die Abhandlung bis in eine gewisse Tiefe. Es ist aber auch möglich, Rekursion ganz zu verbieten oder über Annotationen eine Tiefe vorzugeben.

### 4.2.2 Flexjson

Flexjson ist eine einfache Bibliothek für das Serialisieren und Deserialisieren von JSON-Objekten in Java-Klassen-Objekte.[Fle]

Wenn Attributnamen in JSON von dem Deklarationsnamen im Java-Klassen-Objekt abweichen sollen, müssen Annotationen verwendet werden. Diese Annotationen geben dem Serialisierer Auskunft über die Namen, welche er zu erwarten hat, beziehungsweise welche er schreiben muss.

Beim Serialisieren muss immer explizit angegeben werden, wenn geschachtelte Objekte mit serialisiert werden sollen und natürlich auch wie tief diese verschachtelt werden sollen.

In Programmiererkreisen wird Flexjson für das Debugging eingesetzt. In einer IDE gibt es leider keine Möglichkeit, den aktuellen Stand des Programms zu speichern, geschweige denn diesen Zustand nach gewissen Information zu durchsuchen. [Fle12]

Hier kommt Flexjson ins Spiel, mit dessen Hilfe Entwickler eine Art „Snapshot“, also eine Momentaufnahme der aktuellen Programmwerte, erstellen. Diese „Aufnahme“ kann dann nach Belieben untersucht und gefiltert werden.

### 4.2.3 Jackson

Jackson ist ähnlich wie Flexjson eine Bibliothek für die Serialisierung von Java-Objekten zu JSON-Objekten. Vorteilhaft an Jackson ist, dass die Bibliothek modular aufgebaut ist. So wird für eine einzelne Aufgabe nicht die gesamte Bibliothek benötigt. Auf die verschiedenen Module wird im Kapitel 5 genauer eingegangen.

Vorteilhaft ist die modulare Struktur von Jackson vor allem beim Binden eines Programms. Da nicht zwangsweise alle Module benötigt werden, müssen auch nur benötigte Module in das fertige Programm eingebunden werden, was sich wiederum positiv auf die zu erwartende Größe des Programms auswirkt. [Jac14]

Ein weiterer Vorteil ist, dass es über ein Jackson-Modul möglich ist, Annotations von der *Java Architecture for XML Binding* (JAXB) Bibliothek zu verwenden. Das ermöglicht es, die gleichen Annotations zu nutzen. Dies würde die Bearbeitung und die Übersichtlichkeit sehr vereinfachen, da nicht zwei unterschiedliche Annotations, im Projekt, benutzt werden müssen.

## 4.3 Auswertung der Möglichkeiten

In einer Projektgruppenberatumg wurden alle drei Vorschläge ausführlich erläutert und diskutiert.

Vorteil des eigenen Ansatzes ist es zum einen, dass hier keine fremden Bibliotheken benutzt werden müssen und so keine zusätzlichen „technischen Schulden“ gemacht werden.

In der Informatik ist der Begriff der „technischen Schulden“ eine Anspielung auf Softwarebibliotheken, die schlecht umgesetzt sind und diese schlechte Umsetzung selber ausgeglichen werden muss. Um dies ausschließen zu können, müssten alle genutzen Bestandteile untersucht werden. Da so etwas in der Regel nicht der Fall ist, werden „technische Schulden“ aufgenommen, weil das Gegenteil, also dass die benutzte Bibliothek gut entwickelt wurde, nicht bewiesen ist.

Zum anderen ist es sehr aufwändig, eigene Klassen für die Serialisierung und Deserialisierung zu schreiben, da sehr viele Aspekte beachtet und berücksichtig werden müssen so zum Beispiel die Rekursion oder eine Verschachtelung von Objekt-Instanzen. Auch ein Scanner, der einen geeigneten Objekt-Baum aufbaut, muss entwickelt und implementiert werden. Dies ist im Rahmen dieser Arbeit leider nicht möglich.

Aus diesem Grund schied der eigene Ansatz recht früh aus.

Der zweite Ansatz über die Flexjson-Bibliothek ist interessant, und wurde in der Gruppe lange diskutiert. Denn hiermit ist es möglich, explizit anzugeben, welche Attribute serialisiert werden sollen und welche nicht, wie es vorgesehen ist.

Ein Nachteil von Flexjson ist jedoch, dass es genaue Anweisungen benötigt wann und wie tief es in ein Objekt mit Rekursion oder Verschachtelung hinabsteigt. Dies ist insofern nachteilig, da Quellcode mit Verschachtelung oder Rekursion durch die Annotationen, welcher ohnehin schon schwer zu verstehen ist, noch unleserlicher wird.

Im letzten Ansatz mit Jackson ist es, wie bei JAXB, möglich, den Serialisierer über Annotations zu steuern. Vorteilhaft ist es vor allem, dass der Jackson- und JAXB-Serialisierer die selben Annotations managen kann.

Der große Vorteil der Steuerung über Annotationen ist jedoch auch ein Nachteil, denn diese müssen in allen Klassen angebracht werden, in denen an den Attributen etwas zu beachten ist, wie das sie nicht serialisiert werden sollen. Jedoch sind Annotations nicht zwingend notwendig, wie bei Flexjson zum Beispiel.

Dies war am Schluss auch das entscheidende Element, warum sich für eine Umsetzung mit Jackson und JAXB entschieden wurde. Bei der weiteren Bearbeitung wurden also wenn überhaupt Annotationen von JAXB genutzt.

Diese Arbeit behandelt jedoch nur die Jackson- beziehungsweise JSON-Verarbeitung. In einer anderen Arbeit, die zeitgleich entstand, ist die Verarbeitung mit JAXB und XML zu finden. [Wal14]

### 4.4 Fragestellungen

Wie schon erwähnt, sollen nicht immer alle Attribute serialisiert werden. Ob und wie dies mit Jackson möglich ist, wird im weiteren Verlauf der Arbeit geklärt.

Das ist dann interessant, wenn die Klasse statische Attribute oder Attribute, welche sich im Programmverlauf nicht ändern, enthält. Da diese Attribute immer gleich sind und ihr Ausgangswert bekannt ist, muss dieser nicht serialisiert, versendet oder abgespeichert werden.

Des Weiteren kam die Frage auf, ob es möglich ist, Klassenattribute gesondert zu serialisieren und gegebenenfalls zu untersuchen, wie sich dies auf die Serialisierungs- beziehungsweise Deserialisierungsgeschwindigkeit auswirkt.

Diese Frage ist aufgetreten, da beim Serialisieren gegebenenfalls viele Klassenattribute mit serialiert werden, die nicht mit übertragen, gespeichert oder verwendet werden sollen.

# 5 Jackson und das Jackson-Projekt

Das Jackson-Projekt entwickelt eine freie und modulare Bibliothek, für die Serialisierung und Deserialisierung von Java-Instanzen in JSON-Dokumente und zurück. Jackson wird unter der Contributor License Agreement (CLA) vermarktet. Die zur Zeit aktuelle Version ist 2.4.1, welche auch bei der Bearbeitung des Projektes eingesetzt wird.

## 5.1 Jackson-Module

Die Jackson-Bibliothek besteht aus verschiedenen Modulen, welche wie folgt bezeichnet und die folgenden Aufgabenbereiche haben:

- „jackson-core“, welches die JSON spezifische Implementierung sowie eine Low-Level-Streaming-API enthält
- „jackson-databind“, welches für das *Databinding* verantwortlich ist
- „jackson-annotations“, welches die Jackson spezifischen Annotationen enthält
- „jackson-module-jaxb-annotations“ ist für die Verarbeitung von JAXB-Annotationen verantwortlich
- „jackson-module-jsonSchema“, welches ein JSON-Schema für eine Klasse erstellt

Der Core enthält die Low-Level-Streaming-API, welche die Kommunikation zwischen den einzelnen Modulen übernimmt. Des Weiteren enthält dieses Modul viele Grundklassen, die auch in anderen Modulen benötigt werden. Durch die grundlegenden Klassen und die Aufgabe als Kommunikationsvermittler wird das Modul `jackson-Core` immer benötigt, auch wenn dieses keine Aufgabe der Serialisierung im eigentlichen Sinn übernimmt.

Unter „Databind“ wird eine Methode verstanden, welche über ein Userinterface gesteuert werden kann. Dieses Modul ist in der Lage, Daten aus einem Datenstrom wie zum Beispiel einem JSON-File zu lesen oder zu schreiben.

Das „Databind-Modul“ enthält somit den für JSON notwendigen Scanner und Parser, welche die Umwandlung von Java-Klasse-Objekten in JSON-Objeke und zurück übernehmen.

Die Jackson-Annotations enthalten Informationen, die für das Serialisieren beziehungsweise Deserialisieren verantwortlich sind. Sie sind mit den `AnnotationInspectoren` in diesem Modul zusammengefasst. Jackson kann aber auch Annotations von anderen Serialisierern erkennen und darauf reagieren. Hierfür wird ein weiteres Jackson-Modul benötigt, welches in der Lage ist, die JAXB-Annotations zu verarbeiten (`jackson-module-jaxb-annotations`).

Annotations sind unter Jackson nicht zwingend für eine grundlegende Arbeitsweise des Serialisierers notwendig. Die Annotations werden jedoch für erweiterte Aufgaben wie zum Beispiel das Einschränken des Wertebereichs eines Attributs benötigt.

Mit den drei Hauptmodulen `jackson-core`, `jackson-databind` und `jackson-annotations` ist die Jackson-Bibliothek voll einsetzbar und kann Java-Instanzen zu einem JSON-Datenstrom umwandeln. Der JSON-Datenstrom wiederum kann gespeichert oder an andere Programme gesendet werden.

Um jedoch einheitliche Annotations für Jackson und JAXB zu haben, kommt das eben schon einmal angesprochene Modul `jackson-module-jaxb-annotations` zum Einsatz. Mit dessen Hilfe, Jackson in der Lage ist auch Annotations von JAXB zu verarbeiten.

Eine Mischung von Annotations verschiedener Serialisierer ist grundsätzlich möglich, aber nicht empfehlenswert, da hier die Übersichtlichkeit und die Verständlichkeit des Codes leidet. Aus diesem Grund wird hierauf in der Arbeit nicht weiter eingegangen.

Auch die Erstellung eines JSON-Schemas aus einer Java-Klasse ist mit Hilfe des Moduls `jackson-module-jsonSchema` möglich. Mit Hilfe von den Annotations kann dieses Schema erweitert werden. Die Erweiterungen beziehen sich hauptsächlich auf Werteberiche oder veränderte Namensgebung. Mit Annotations, welche für die Namensgebung verantwortlich sind, ist es möglich, im JSON-Dokument andere Namen zu verwenden als die Attributnamen. Wie eine Schemaerzeugung genau funktioniert, wird im Kapitel 5.6 genauer erklärt. [Jac14]

## 5.2 Serialisierung mit Jackson

Der im folgenden beschriebene Sachverhalt erläutert das Quellcode-Listing auf der nächsten Seite.

Um eine Serialisierung mit Jackson umzusetzen, wird zuerst eine Instanz der Klasse `ObjectMapper` benötigt, welche den Databinder darstellt, welcher, wie schon dargestellt, den Datenstrom verarbeitet. Dieser `mapper` ist somit für die Konvertierung von Java-Instanzen zu JSON-Dokumenten verantwortlich. Das zu serialisierende Objekt ist im Beispiel `opmObject`, welches der Methode `serialize` übergeben wird.

Jedoch wird nicht nur der `mapper` benötigt, sondern auch ein `AnnotationInspector`, der jedoch abstrakt ist. Der `inspector` wird deshalb als Instanz von `JaxbAnnotationInspector` erstellt, was durch eine Implementierung der abstrakten Klasse möglich ist. Weiterführende Überlegungen zum `AnnotationInspector` sind im Kapitel 5.2.1 näher erläutert.

Dem `inspector` wird eine `TypeFactory` mit „Default-Einstellungen“ übergeben . Dies bedeutet, es wird auf die Original JAXB-Annotations geparst, ohne auf Sonderfälle zu achten. Andere Annotations werden nicht berücksichtigt. Der `inspector` wird schließlich, nach der Erstellung, dem `mapper` übergeben, damit dieser auf die entsprechnden Annotations reagieren kann.

Um eine *Minimal Exception Safety* zu garantieren wird nun eine `null`-Abfrage des zu serialisierenden Elements (`opmObject`) gemacht. Mit dieser Stufe der Sicherheit soll nicht verhindert werden, dass eine Exception passiert. Es wird lediglich garantiert, dass die Methode ohne abzustürzen durchlaufen werden kann. [Gri02]

Ist die zu serialisierende Instanz `null`, so wird eine `IllegalagumentException` generiert und die Methode so ordnungsgemäß beendet. Ist eine Instanz vorhanden, wird diese dem `mapper` übergeben. Das Ergebnis des Aufrufs der Methode `writeValueAsString` von der Klasse `ObjectMapper` ist entweder bei Erfolg ein valider JSON-String oder beim Scheitern eine `JsonProcessingException`.

Eine `JsonProcessingException` wird generiert, wenn Probleme beim parsen, beziehungsweise beim Generieren des JSON-Kontent auftreten, die keine reinen I/O-Probleme sind. Die Exception erbt jedoch von `IOExecption`. Bei der `JsonProcessingException` handelt es sich um eine „checked“ Exception welche irgendwo im Programmablauf abgefangen werden muss.

Über den Methodenaufruf `writeValueAsString` wird also der Serialisierungsprozess mit der Jackson-Bibliothek in Gang gesetzt.

Aus Gründen der Verständlichkeit, wurde die `null`-Abfrage nicht am Methodenanfang gemacht, um bei einer Fehleingabe die Methode frühestmöglich zu verlassen, ohne vorher Objekte zu erstellen.

```
public String serialize(OPMObject opmObject) {
    ObjectMapper mapper = new ObjectMapper();
    AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(
        TypeFactory.defaultInstance());
    mapper.setAnnotationIntrospector(inspector);

    if (opmObject == null) {
            throw new IllegalArgumentException("OPMObject can not be null!");
    }
    try {
        return mapper.writeValueAsString(opmObject);
    } catch (JsonProcessingException e) {
        e.printStackTrace();
        return null;
    }
}
```

### 5.2.1 AnnotationInspector

Da wie eben schon einmal erklärt, der `AnnotationInspector` als abstrakte Klasse implementiert wurde, muss diese mit einer implementierten Instanz instanziiert werden. Dies kann wie im Kapitel 5.2 mit dem `JaxbAnnotationInspector` oder einem anderen `AnnotationInspector` geschehen.

Durch die Abstraktion der Klasse `AnnotationInspector` ist es möglich, hier weitere Inspektoren zu implementieren.

So wäre es zum Beispiel möglich, einen `SMDAnnotationInspector` selber zu entwickeln, welcher nicht auf Annotations in einer Klasse achtet, sondern auf die im Projekt vorhandenen SMD zurückgreift.

Mit dieser Lösung müssten keine Annotations an die eigentliche Klasse angebracht werden. Diese Informationen könnten durch den `SMDAnnotationInspector` direkt über den SMD-Assistenten und der SMD-Datenbank geladen werden. Eine genauere Beschreibung zum SMD-Assistenten ist im Kapitel 6 zu finden.

## 5.3 Deserialisierung mit Jackson

Der im folgenden beschriebene Zusammenhang ist noch einmal im Quellcode-Listing auf der nächsten Seite zu finden.

Für die Deserialisierung mit Hilfe von Jackson wird wie bei der Serialisierung ebenfalls ein Databinder, also ein Objekt der Klasse `ObjectMapper` und ein `AnnotationInspector` benötigt, welche wie im Kapitel 5.2 schon beschrieben, erstellt werden.

Bevor dies jedoch passiert, wird geprüft, ob der eingegebene String weder `null` noch `empty` ist, womit wieder die *Minimal Exception Safety* garantiert werden kann. Sollte einer dieser Fälle auftreten, so wird eine `IllegalArgumentException` zurückgeliefert und die Methode ordnungsgemäß verlassen.

Wenn der String, wie eigentlich zu erwarten ist, einen Inhalt hat, wird die Methode `readValue` vom `mapper` mit dem übergebenden String und der Information, um welche Klassen-Instanz es sich beim String handelt, übergeben. Der zurückgegebene Typ, der Methode `readValue`, entspricht dem Typ der im zweiten Argument übergebenen Instanz. Der Methodenaufruf `readValue` setzt somit unter Jackson die Deserialisierung in Gang.

Die Schwierigkeit beim Deserialisieren besteht nun darin, dass bevor der String überhaupt deserialisiert werden kann, erst festgestellt werden muss, um welche Klasse es sich eigentlich handelt.

Im Codebeispiel unten wird momentan noch davon ausgegangen, dass es sich immer um eine Instanz der Klasse „TestData“ handelt. Wie diese Einschänkung aufgehoben werden kann, wird im folgenden Kapitel genauer beschrieben.

```
public <T extends OPMObject> T deserialize(String string) {
        if (string == null) {
            throw new IllegalArgumentException("String can not be null!");
        }
        if (string.isEmpty()) {
            throw new IllegalArgumentException("String can not be empty!");
        }

        ObjectMapper mapper = new ObjectMapper();
        AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(
            TypeFactory.defaultInstance());
        mapper.setAnnotationIntrospector(inspector);
        try {
            return (T) (mapper.readValue(string, TestData.class)));
        } catch (IOException | ClassNotFoundException | ClassCastException e) {
            e.printStackTrace();
        }
        return null;
}
```

## 5.4 Instanzunabhängige Deserialisierung

Wie gerade schon erkläutert, wird beim Deserialisieren vorausgesetzt, dass der Typ der Instanz des zu deserialisierenden Strings bekannt ist. Um den String nun eindeutig einem Typ zuzuordnen, musste eine eindeutige Kennzeichnung geschaffen werden.

Es wurde daraufhin in der Projektgruppe eine Einigung darüber getroffen, dass der Klassenname über ein String-Attribut `className` hinzugefügt wird. Aus diesem Grund bekam die Klasse `OPMObject` ein String-Attribut `className`, in welchem der Klassenname der jeweiligen Klasse abgelegt ist.

Durch die gemeinsame Wurzelklasse `OPMObject` haben nun alle Klassen das Attribut `className` geerbt. Daraus ergibt sich wiederum im Umkehrschluss, dass nur noch Klassen, welche von `OPMObject` erben, serialisiert werden können, da nur sie mit Sicherheit das `className`-Attribut enthalten. Beim Serialisieren wird nun der Klassenname mit in den Ausgabe-String geschrieben und dieser kann dadurch eindeutig identifiziert werden.

Um nun den Klassennamen aus dem String zu lesen, wurde die neue Methode `getClassFileFromString` geschrieben, welche den Klassennamen aus dem String filtert und diesen dann zurückliefert. Das Auslesen des Typ-Namen wurde mit Hilfe der `split`-Methode der String-Klasse realisiert.

Diese Umsetzung ist vorläufig, da sie nicht effizient genug arbeitet, jedoch wesentlich schneller als Jacksons eigenes Parsen auf einen Schlüsselnamen ist.[Mky11]

### 5.4.1 JSON-Parser

Über einen zusätzlichen JSON-Parser könnte der Effizenznachteil einer `split`-Methode, oder des Jackson-Parsers aufgehoben werden. Hierfür gibt es unter Java verschiedene Möglichkeiten. So wäre eine Einbindung von `org.json`, `Gson`, `JSON.simple` oder `minimal-json` möglich.[Ste13]

Aus zeitlichen Gründen, konnten keine Untersuchungen zu weiteren JSON-Parsern gemacht werden. Jedoch wurde im Zuge der instanzunabhängigen Deserialisierung der Jackson-Parser getestet und für untauglich befunden, da er mindestens die zehnfache Zeit der `split`-Methode benötigt.

## 5.5 Klassendiagramm der Serialisierung

Wie von OPM verlangt, erben hier alle Klassen von `OPMObject`. Um die Geschwindigkeit, beziehungsweise die Funktionalität der XML- und JSON-Serialisierung vergleichen zu können, wurde eine gemeinsame abstrakte Klasse `Serializer` erstellt.

Die Klasse `JSONSerializer` erbt, um einen direkten Vergleich durchführen zu können, genau wie `XMLSerializer` von `Serializer`. `Testdata` und `TestData2OPM` sind erste Test-Klassen von denen Instanzen serialisiert und deserialisiert werden. Diese werden im späteren Verlauf des Projektes durch sinnvolle Testklassen ersetzt und sind später nicht mehr aufzufinden.

Main-Klasse in diesem Projekt ist `OPM_Serializer`. Die `main`-Methode setzt die Serialisierung zum Testen in Gang. Hierbei werden die Testklassen und die Serialisierer deklariert und instanziiert, welche für die Tests benötigt werden.

Damit, wie gewünscht, jede Klasse serialisiert werden kann, wurde die Methode `serializeMe` zu `OPMObject` hinzugefügt. Der Methode muss eine Instanz des Serialisierers übergeben werden, welcher für die Serialisierung genutzt werden soll.

Durch die Vererbung im OPM-Modell, erbt jede Klasse die Methode `serializeMe`. Diese Methode nutzt nun bei Aufruf die `serialize`-Methode des jeweiligen Serialisierers, um die Serialisierung durchzuführen.

Einen vollständigen Überblick gibt das unten gezeigte Klassediagramm.

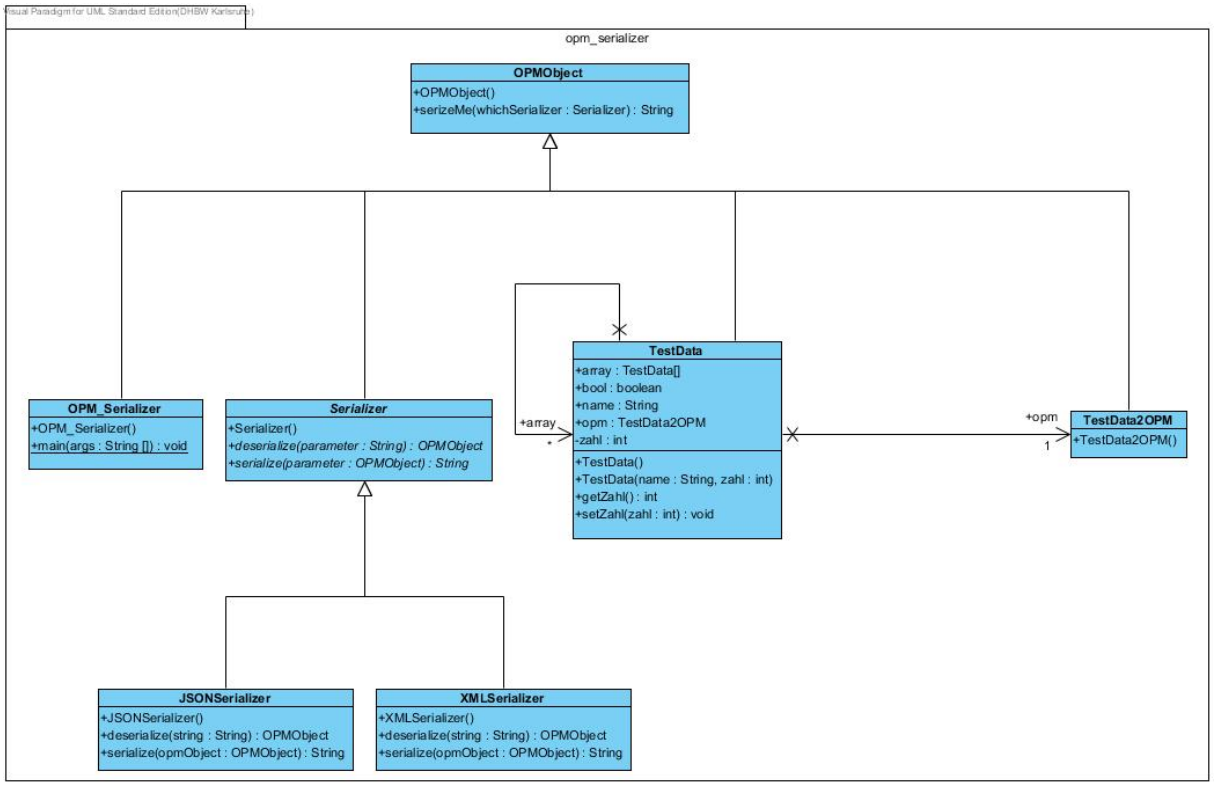


Abbildung 1: Klassendiagramm der Serialisierung

## 5.6 JSON-Schema-Erstellung aus einer Klasse

Ähnlich wie bei XML gibt auch es in JSON ein Datenformat, welches die zu erwartende Form des Streams vorgibt, das JSON-Schema. Wie in einer XSD, wird das Schema in der eigenen Norm als konformes Dokument erstellt. [JSO14]

Ein Beispiel für ein JSON-Schema wird hier anhand des JSON-Dokuments in Kapitel 3.1 dargestellt. Das JSON-Schema besteht aus einem Objekt, welches die jeweiligen Namen und Werte der Attribute einer Instanz enthält und ist beispielhaft am Ende des Kapitels zu finden.

Somit steht am Anfang eines JSON-Schemas immer der Bezeichner `type` mit dem entsprechenden Wert, nämlich `object`. Dies ist zwingend erforderlich, da jedes JSON-Dokument auch eine JSON-Objekt ist. Unter dem Schlüssel `properties` sind, wiederum als inneres Objekt, die Attribute mit ihren möglichen Ausprägungen aufgelistet.

Es gibt natürlich weitere Bezeichner (ähnlich wie `type`), die den Wertebereich des Attributs weiter eingrenzen. Sie werden jedoch erst angezeigt, wenn die entsprechenden Annotations im Quellcode angebracht sind. Auf weitere Bezeichner wird im folgenden jedoch nicht weiter eingegangen, da dieses Thema zu umfangreich ist, um es in dieser Arbeit weiter zu vertiefen.

Am Anfang einer Property steht der Name des Atrributs, (im Beispiel `"stringValue"`), wie dargestellt. Gefolgt wird dieser Name von einem Objekt, welches die genauen Eigenschaften des Attributs beschreibt.

Um die XML-Annotationen von JAXB bei der Schemaerzeugung zu nutzen, muss wieder ein `AnnotationInspector` verwendet werden. In diesem Fall ist das der `JaxbAnnotationInspector`.

Die Einschränkungen und Beschreibungen der Attribute werden dem Serializer mit Hilfe der Annotationen übergeben, welche Java-Methodennamen enthalten. Das Setzen der Beschreibungen geschieht über Java-Methoden, welche zur Laufzeit vom Serializer aufgerufen werden.

```
{
    "type":"object",
    "proterties":{
        "stringValue":{
            "type":"string"
        },
        "boolValue":{
            "type":"boolean"
        },
        "nullValue":{
            "type":"string"
        },
        "zahlValue":{
            "type":"integer"
        },
        "fliesskommaValue":{
            "type":"integer"
        },
        "arrayValue":{
            "type":"array",
            "items":{
                "type":"string"
            }
        },
        "objectValue":{
            "type":"object",
            "properties":{
                "zahl_2Value":"integer"
            }
        }
    }
}
```

## 5.7 JSON-Schema mit Hilfe von Jackson erstellen

Seit der Jackson Version 2.2 wurde das Modul zur Erstellung eines JSON-Schemas („`jackson-modul-jsonSchema`“) aus dem Modul „`jackson-databind`“ ausgegliedert und ein eigenes Modul mit erweitertem Funktionsumfang eingeführt.

Für die Erstellung eines JSON-Schemas wird somit das Zusatzmodul „`jackson-module-jsonSchema`“ benötigt. Um unnötige Fehlerquellen zu vermeiden, wurde auch dieses Modul in der Version 2.4.1 verwendet, auch wenn es zum Erstellungsdatum schon eine neuere Version gab. Somit sind alle Module von der selben Version und Fehler durch Versionsunterschiede sind ausgeschlossen.

Im Codebeispiel unten wird der nun folgende Zusammenhang noch einmal verdeutlicht.

Wie schon in früheren Beispielen gezeigt, wird zuerst wieder ein `ObjectMapper` erstellt. Hierbei kann eine `JsonMappingException` auftreten, welche entweder weitergereicht werden kann oder direkt behandelt wird. Da die Exception „checked“ ist, muss sie auf jeden Fall irgendwo im Programm behandelt werden.

Des Weiteren wird für die Schemaerzeugung noch ein `SchemaFactoryWrapper` erstellt, welcher das Schema erstellt. Der Schema-Wrapper kann unter Umständen die `JsonProcessingException` werfen, welche im Kapitel 5.2 genauer erläutert wird.

Der Wrapper wird nun, über den nächsten Methodenaufruf, auf den Mapper und somit auf die entsprechend zu mappende Instanz angesetzt. Dies geschieht über die Methode `acceptJsonFormatVisitor`.

Mittels der Methode `finalSchema` wird das Schema der Java-Klasse als `JsonSchema` erstellt.

Über den `return`-Wert wird das `JsonSchema` als String übergeben.

```
1  public String generateSchema(OPMObject opmObject) throws JsonProcessingException
       ,JsonMappingException{
2          ObjectMapper m = new ObjectMapper();
3          SchemaFactoryWrapper visitor = new SchemaFactoryWrapper();
4
5          m.acceptJsonFormatVisitor(m.constructType(opmObject.getClass()),
6              visitor);
7          JsonSchema jsonSchema = visitor.finalSchema();
8
9          return m.writerWithDefaultPrettyPrinter().
10             writeValueAsString(jsonSchema);
11 }
```

## 5.8 Auffälligkeiten beim Testen

Beim Serialisieren und Deserialisieren sollen die Attributwerte einer Klasse in JSON-Dokumenten gespeichert, beziehungsweise JSON-Objekte in Attributwerte gewandelt werden. Hiefür benötigen alle Klassen einen Standard-Konstruktor, damit der Serializer diesen beim Erstellen einer Klasseninstanz aufrufen kann, um eine „neue“ Instanz zu erstellen.

Neu bedeutet in diesem Kontext, dass die Instanz zwar schon einmal existierte, aber zur jetzigen Laufzeit erst erstellt werden muss.

Des Weiteren benötigt der Serializer für alle nicht `public` Attribute Getter- und Settermethoden, um Zugriff auf diese Attribute zu erhalten. Denn der Serializer darf durch Java-Richtlinien nur auf `public` Attribute ohne Getter- und Settermethoden zugreifen. Jedoch sollen nach OPM keine Attribute `public` sein.

Um eine Serialisierung von allen Klassen zu gewährleisten, muss das OPM-Modell angepasst werden. Denn laut OPM sind Getter- und Settermethoden nicht zwingend notwendig und nur

optional. Da diese Methoden aber für die Serialisierung notwendig sind, wurde OPM angepasst und nun sind Getter- und Settermethoden Vorraussetzung.

Da alle Klassen sich an die OPM-Regeln halten, ist somit wieder gewährleistet, dass alle ankommenden Instanzen serialisiert oder deserialisiert werden können.

In ersten Tests mit dem `JSONSerializer` wurde die Funktionsfähigkeit bewiesen.

Es war nicht möglich Klassen-Objekte getrennt von der anderen Klasse über den Serialisierer zu trennen. Jedoch kann eine Trennung der Objekte auch manuell nach der Serialisierung vorgenommen werden.

### 5.8.1 Auffälligkeiten beim Erstellen des JSON-Schemas

Nach einigen Tests, wurde festgestellt, dass Jackson für alle Zahlen immer `integer` im Schema angibt, obwohl JSON auch `double` oder `float` kennt. Dabei ist es egal, ob es sich in der Klasse um `float, double` oder `integer` handelt. Warum dies im Schema nicht ordnungsgemäß übernommen wird, konnte trotz Recherche nicht herausgefunden werden.

# 6 Der SMD-Assistent im GDS-System

Die SMD enthalten alle wichtigen Informationen um die Struktur einer Klasse zu beschreiben. Diese Klasseninformationen sollen später von einem SMD-Assistenten verwaltet werden.

Die SMD-grundlegenden Informationen wie Klassenname, Methodennamen und Attributnamen, aber auch Attribut-Qualifier und -Modifier, sowie Methoden-Qualifier, -Modifier und Parameterlisten der Methoden, sind vorhanden. Zusätzlich ist gespeichert, von welcher Klasse geerbt wird, beziehungsweise welche implementiert wurden.

In einer MySQL-Datenbank können alle eben genannten Klasseninformationen gespeichert werden. Zusätzlich werden in der Datenbank Referenzen zu Klassen aufgebaut, von welchen geerbt beziehungsweise welche implementiert wurden. Außerdem werden Klassenattribute in der Datenbank referenziert. [Zil14]

Da es sich bei den SMD um reine Metadaten handelt, werden keine Methodenrümpfe in die SMD-Datenbank aufgenommen.

Der Nutzer des GDS-Systems gibt über den *User Data Description Editor* (UDDE), also das User Interface, seine Klassen und *Anwendermetadaten* (AMD) in das System. Er kann über den UDDE ein komplettes „Klassengrüst“, also alle SMD erstellen, indem er alle Klassenattribute und Methoden angibt, sowie referenzierte Klassen angibt.

Hierbei ist er natürlich auch in der Lage den Modifier und den Rückgabetyp zu bestimmen. Bei Methoden können natürlich zusätzlich Parameterlisten angegeben werden.

Des Weiteren legt der UDDE die generierten „strukturellen Metadaten“, mit Hilfe des SMD-Assistenten in einer Datenbank ab, welche später vom SMD-Assistenten auch wieder ausgelesen werden können.

Der SMD-Assistent ist für das Umwandeln von Klassen in SMD verantwortlich. Nach dem Erstellen der SMD werden diese zur Aufbewahrung vom Assistenten in einer Datenbank verwaltet.

Die SMD werden von der Anwendung an verschiedenen Stellen benötigt, so zum Beispiel beim Serialisieren. Ein *Class Generator* (CG) wandelt die SMD wieder in Programmcode, welcher OPM-konform erzeugt wird. Eine genaue Spezifikation für den Klassengenerator ist im Kapitel 6.2 zu finden. Die generierten Daten können dann ebenfalls vom GDS in einer Datenbank gespeichert werden.

Der *Interface Generator* (IG) erstellt aus den vom SMD-Assistenten gelieferten SMD Schemata für JSON und XML. Eine Funktion zur Erstellung des JSON-Schemas aus einer Klasse wird in Kapitel 5.6 beschrieben. Auch das entsprechende Klassenschema soll vom GDS in einer Datenbank abgelegt werden.

Der beschriebene Zusammenhang der Komponenten des GDS ist im Bild auf der nächsten Seite noch einmal verdeutlicht.

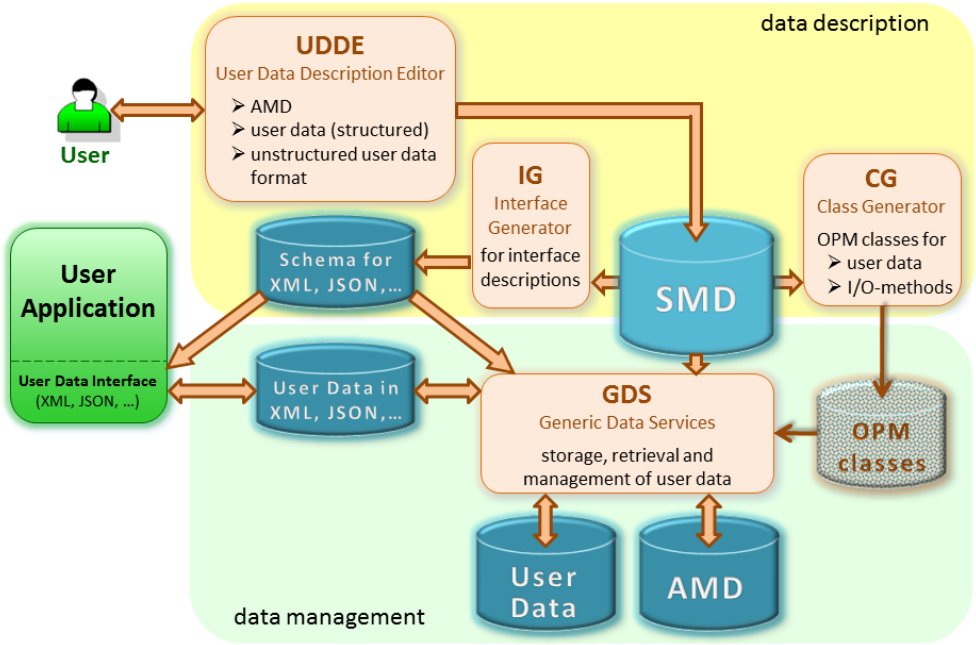


Abbildung 2: GDS Übersicht

## 6.1 Spezifikation des SMD-Assistenten

Im Verlauf der Arbeit wurde zunehmend klar, dass die Spezifikation des SMD-Assistenten nötig wird, da die SMD, wie im Schaubild oben zu erkennen, zentraler Bestandteil des Projektes sind. Der SMD-Assistent wurde im Projekt öfter diskutiert und seine Funktionsweise soll an dieser Stelle einmal genauer erläutert werden.

### 6.1.1 Funktionen des SMD-Assistenten

Der SMD-Assistent soll die „strukturellen Metadaten“ aus der Datenbank laden und diese an das GDS, den IG oder den CG weiterreichen. Somit ruft jede Instanz, welche die SMD benötigt, den SMD-Assistenten auf, damit dieser die benötigten Informationen liefern kann.

Zu einer `ObjectID`, einer `ClassID` oder einem Klassennamen müssen die passenden Metadaten aus der Datenbank geladen werden. Die geladenen Daten werden in einer Instanz der Klasse `ClassDecr` zusammengefasst und mittels dieser Instanz übergeben.

Objekte, welche serialisiert werden können, haben eine `ObjectID`, anhand welcher sie eindeutig identifiziert werden können. Zusätzlich können über die `ObjectID` auch verschiedene Instanzen eines Objektes unterschieden und zugeordnet werden. Dies geschieht über die in der `ObjectID` vorhandene `InstanzID`.

Mittels einer `ClassID` können genau wie mit der `ObjectID` Objekte identifiziert werden, jedoch keine bestimmten Instanzen.

Eine Instanz der Klasse `ClassDecr` enthält mittels der Klassen `MethodDescr` und `AttrDecr` alle nötigen Informationen, um eine Klasse rekonstruieren zu können. Sie stellen die „strukturellen Metadaten“ dar.

`MethodDescr` enthält eine Methode der Klasse, mit einer Liste von allen Parametern und deren Typen. Die Klasse `AttrDecr` hingegen hält jeweils ein Attribut mit dessen Informationen wie zum Beispiel Typ und Modifier. [Zil14]

Der Zusammenhang ist im Klassendiagramm der SMD-Klassen noch einmal dargestellt. Im Bild 3, auf der nächsten Seite, ist aus Gründen der Übersichtlichkeit nicht der gesamte OPM-Klassenbaum abgebildet, sondern lediglich die für den SMD-Assistent wichtigen Klassen sind aufgeführt.

Einmal aus der Datenbank geladene SMD soll der SMD-Assistent zwischenspeichern, um beim erneuten Abfragen schneller reagieren zu können.

Damit es nicht zu einem Arbeitsspeicherüberlauf kommt, muss der SMD-Assistent genau wie zukünftige andere Assistenten auch eine Möglichkeit besitzen, seinen internen Cache zu verkleinern. Dies soll über einen möglichst effizienten Scheduling-Algorithmus geschehen, welcher implementiert und auf Effizienz geprüft werden muss.

Hier kommt ein weiterer Assistent zum Einsatz, und zwar der Speicher-Assistent, welcher bei geringem Arbeitsspeicher einen Befehl an alle Assistenten schickt, damit diese ihren Speicherbedarf reduzieren. Durch die Serialisierer wurde die Überlegung zum Speicher-Assistenten zum ersten Mal entfacht, da hier teilweise sehr viel Arbeitsspeicher benötigt wird. Dazu aber im nächsten Kapitel mehr.

### 6.1.2 Der SMD-Assistent unter Java

Unter Java kann die Arbeitsweise des SMD-Assistent deutlich vereinfacht werden, da hier die SMD nur bei explizitem Verlangen geliefert werden müssen, denn Java verwendet mit den *.class*-Objekten eigene Metadaten, über die eine Verarbeitung unter Java einfacher umgesetzt werden kann. Der SMD-Assistent sollte unter Java in der Lage, sein direkt Klassenobjekte zu liefern, da sich diese schneller verarbeiten lassen als die SMD-Daten.

## 6.2 Der Klassengenerator

Der Klassengenerator ist ein Modul des GDS, welches aus den vom SMD-Assistenten gegebenen strukturellen Metadaten wieder Programmcode generiert. Es soll eine abstrakte Klasse `ClassGenerator` geben, welche abstrakte Methoden, zum Bauen einer Klasse, zur Verfügung stellt.

Für jede Programmiersprache, soll nun ein Klassengenerator von der abstrakten Oberklasse abgeleitet werden. Die programmiersprachenspezifischen Klassengeneratoren sind dann in der Lage, für ihre Programmiersprache, aus den SMD, Klassen zu generieren.

Da die Serialisierer wie Jackson und JAXB für Attribute, die nicht `public` sind, Getter- und Setter-Methoden benötigen, muss der Klassengenerator diese erzeugen, auch wenn diese nicht in den SMD auftauchen. Des Weiteren muss er diese Methoden auch nach dem OPM-Standard implementieren. Dies bedeutet, das in Setter-Methoden die Attribute gesetzt und in Getter-Methoden die Attribute gelesen werden müssen.

Außerdem benötigen die Serialisierer auch Standardkonstruktoren, um ein leeres Element zu erstellen, welches im Laufe der Deserialisierung gefüllt werden kann.

Um die Serialisierer steuern zu können, sind gegebenenfalls Annotations an den Klassen notwendig, welche ebenfalls vom `ClassGenerator` angebracht werden müssen. Eine andere Möglichkeit wäre, wie schon im Kapitel 5.2.1 erwähnt, die Verwendung der SMD als Annotationsersatz.

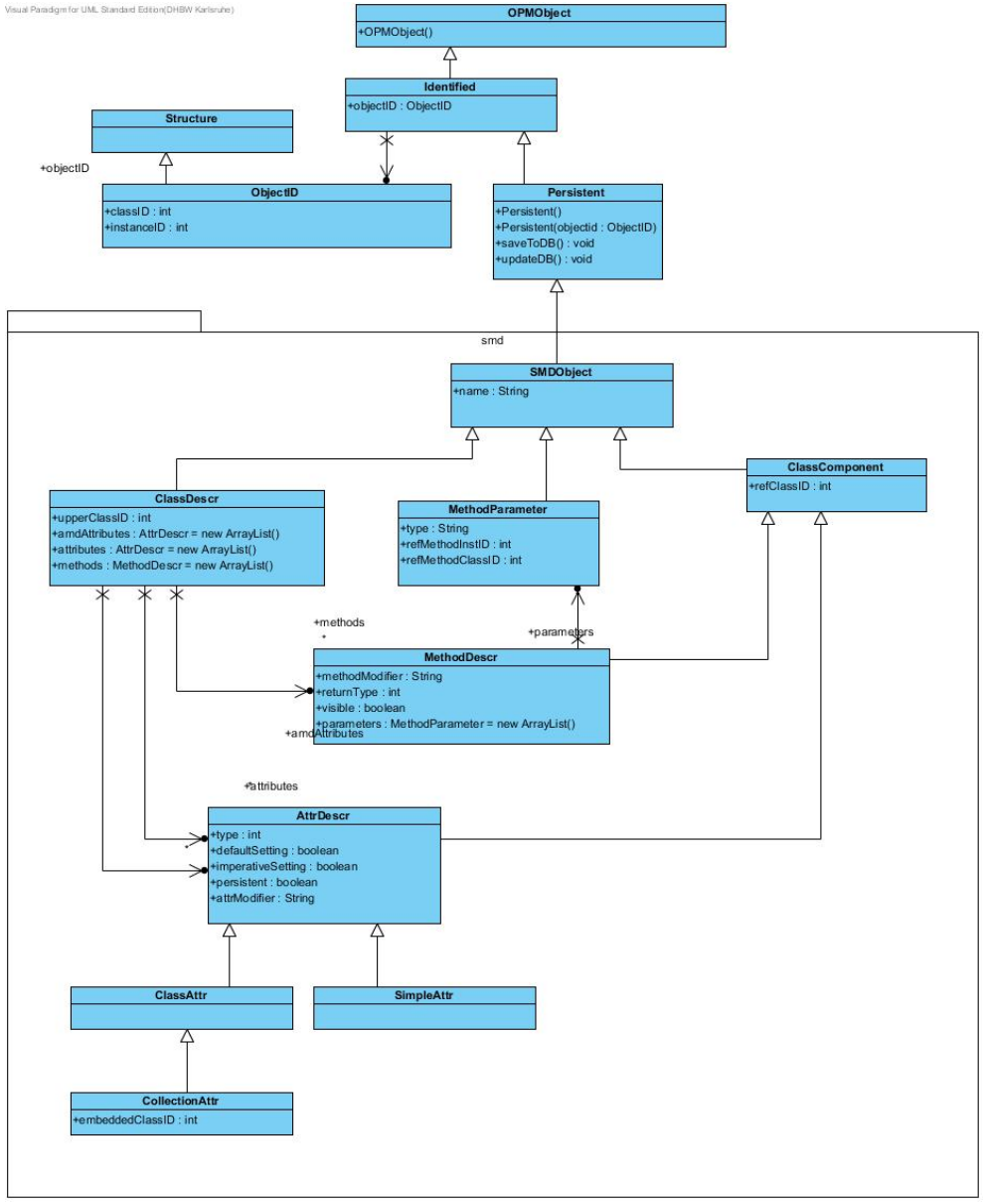


Abbildung 3: Klassendiagramm der SMD-Klassen

# 7 Test des JSON-Serialisierers

Im Verlauf dieses Kapitels wird die benötigte Arbeitszeit zum Serialisieren und Deserialisieren, sowie das Verhalten von Jackson genauer untersucht und eine Auswertung dieser Tests vorgenommen.

## 7.1 Testaufbau

Im Test wird der Quellcode, wie im Verlauf der Arbeit beschrieben, eingesetzt, um das Verhalten der JSON-Bilbliothek genauer zu beurteilen. Hierfür wurden die beiden vorläufigen Testklassen wurden durch sinnvolle Testklassen ersetzt .

In den folgenden Unterkapiteln zum Testaufbau ist immer nur von Serialisierung die Rede, was aus Übersichtlichkeitsgründen getan wurde. Gemeint ist jedoch nicht nur die Serialisierung, sondern auch die Deserialisierung.

## 7.2 Testaufbau Software

Der Test wird mit speziell für Testzwecke geschriebene Klassen durchgeführt, wobei es sich um fünf Testklassen handelt.

Die erste Klasse enthält genau einen `short`-Wert als Attribut zur Serialisierung. Mit Hilfe dieser Klasse soll, unter Beachtung der anderen Tests, herausgefunden werden, wie lange der Anlauf und das Ausführen einer Serialisierung benötigt. Diese Aussage ist möglich, da abgesehen von zwei Byte keine weiteren Daten der Klasse serialisiert werden und somit sollte die Bearbeitung eines Attributs keine Auswirkung haben.

Eine Klasse, welche genau ein Gibibyte ($2^{30}$ = 1 073 741 824 Byte) an Daten beinhaltet, soll etwas über die Zeit für die Serialisierung von einem Element aussagen, da hier die Zeit für das Anlaufen des Serialisieres vernachlässigt werden kann. Um die Datenmenge zu erreichen wird, eine Klasse mit 536 870 912 `short`-Werten in einem Array erzeugt, denn jeder `short`-Wert ist 2 Byte groß, wodurch sich die genaue Größe von einem Gibibyte ergibt. Die `short`-Werte werden im Konstruktor bei Erstellung der Klasse zufällig erzeugt.

Die dritte Testklasse enthält von jedem möglichen „Standard-Datentyp“ genau ein Attribut mit einem Wert. Sie dient hauptsächlich als Vergleichspunkt zu den beiden noch fehlenden Klassen, aber auch um zu beweisen, dass jeder Datentyp serialisiert werden kann.

Eine weitere Klasse ist genau wie die dritte beschriebene Vergleichsklasse aufgebaut, jedoch mit dem Unterschied, dass diese ein weiteres Attribut besitzt. Bei dem neuen Attribut, handelt es sich um ein Klassenattribut, welches der dritten, beschriebenen, Klasse entspricht. Geprüft werden soll nun, ob die Serialisierung ähnlich lange wie bei der dritten Klasse dauert. Sollte das der Fall sein beinhaltet der eingesetzte Serialisierer einen Cache, bzw. arbeitet vorausschauend.

Bei der letzten zu testenden Klasse handelt es sich um eine rekursive Version der dritten Klasse, wobei zwei Rekursionsstufen gemacht werden. Hier soll geprüft werden, wie der Serialisierer mit Rekursion umgeht, und welche Zeit er hierfür beansprucht.

## 7.3 Testaufbau Hardware

Die Tests wurden auf einem Arbeitsplatzrechner mit Kubuntu 12.04.2 Linux 3.2.0.58-generic x86-64 durchgeführt.

Die Hardwaredaten des Rechners waren folgende:

- CPU: 2x Intel Xeon 5148 mit 2,33 GHz
- RAM: 8GB DDR-2 667 MHz (Zugriffszeit 1,5 ns)
- Chipsatz: Intel 5000 Series Chipset
- Festplatte: Seagate ST31000528AS (1 TB) davon 16 GB Swap

Zur Festplatte sollte noch gesagt sein, das sich das Betriebssystem, Swap und Datenpartition jeweils auf getrennten Partitionen befinden, jedoch physisch auf einer Festplatte sind.

Bei allen Tests wurde darauf geachtet, dass der Swap nicht benutzt wird, um Verzerrungen der Messzeit zu verhindern.

## 7.4 Testdurchführung

Die Testdurchführung wurde zuerst zusammenhängend ausgeführt, dass heißt, alle Tests wurden automatisiert vom Programm ausgeführt. Jedoch wurden sehr früh Effekte verschiedener Caching-Mechanismen festgestellt, was eine genaue Messung der Tests auf diese Art nicht möglich macht.

Zum einen werden bei der automatischen Testausführung Hardware-Caches verwendet, zum anderen ist der JIT-Compiler in der Lage, diesen Testablauf weiter zu optimieren. Um dies zu verhindern, wurde wie folgt vorgegangen.

Es wurde somit entschieden, jeden Test einzeln durchzuführen und ihn jeweils dreimal zu wiederholen, wenn keine extremen Schwankungen auftreten. Sollte das der Fall sein muss ein Test öfters wiederholt werden, bis die Testzeit eindeutig ist.

### 7.4.1 Auffälligkeiten bei Testbeginn

Die größte Klasse mit einem Gibibyte Daten bereitetete Probleme. Jackson ist nicht in der Lage, so viele Elemente in einer Instanz zu serialisieren.

Dies liegt daran, dass bei einer solch großen Datenmenge extrem große Strings entstehen, welche die `StringBuffer`-Klasse, die intern in Jackson verwendet wird, nicht verarbeiten kann, da diese auf Arrays bassiert, welche die Größe beschränken.

Da dieser Effekt auch bei fünfhundert Mebibyte auftrat, wurde entschieden, die große Klasse auf zweihundertfünfzig Mebibyte zu beschränken. Daraus resultiert, dass sich nur noch 134 217 728 `short`-Werte im Array befinden. Was eine Datengröße von 256 Mebibyte darstellt.

Dieser Effekt, zeigt sich jedoch nur bei der `jre-1.7.0.60 Open-JDK`. Das Java Runtime Environment der Firma Oracle funktioniert genau wie die `jre-1.7.0.45 Open-JDK-Bibliothek` bis zu einer Klassengröße von 512 Gibibyte fehlerfrei.

## 7.5 Testergebnisse

Die folgenden Testergebnisse wurden wie oben beschrieben ermittelt, wobei die Endzeit in Millisekunden des Vorgangs von der Startzeit in Millisekunden angezogen wurde.

Das Serialisieren einer Klasse mit einem `short`-Wert dauerte bei drei Durchgängen 338, 335 und 337 Millisekunden. Die Deserialisierung betrug im Gegensatz dazu wesentlich weniger Zeit, nämlich 40, 40 und 41 Millisekunden.

Die Klasse mit allen möglichen Datentypen benötigte für die Serialisierung 355, 344 und 346 Millisekunden. Das Deserialisieren benötigte kaum mehr Zeit als bei nur einem Wert und zwar 46, 46 und 50 Milisekunden.

Ein Klassenattribut in einer Klasse benötigte für das Serialisieren 355, 341 und 342 Millisekunden. Die Deserialisierung dauerte 49, 50 und 49 Millisekunden

Derselbe Dateninhalt, jedoch rekursiv aufgerufen, brauchte zum Serialisieren 343, 339 und 347 Millisekunden. Beim Deserialisieren ergaben sich 47, 47 und 48 Millisekunden.

Das Serialisieren der großen Klasse benötigte 10632, 10751 und 10521 Millisekunden. Zum Deserialisieren wurde eine Zeit von 17012, 16836 und 17127 Millisekunden benötigt.

## 7.6 Auswertung der Testergebnisse

Aus der Serialisierung der großen Klasse lässt sich sehr gut entnehmen, dass die Serialisierung eines einzelnen Elements quasi keine Zeit in Anspruch nimmt, denn die Zeit für ein Element liegt bei $7,67111 * 10^{-5}$ Millisekunden.

Über Jackson kann also ausgesagt werden, dass das Starten des Serialisierungsprozesses in etwa 336 Millisekunden dauert, da das Serialisieren einer Klasse mit einem Attribut im Mittel diese Zeit benötigt und somit die Serialisierung dieses Attributs vernachlässigt werden kann, da es quasi keine Zeit in Anspruch nimmt, wie die Serialisierung der großen Klasse gezeigt hat.

Auch durch die folgenden Ergebnisse ist diese Aussage möglich. Da auch eine Klasse mit mehreren Elementen (Klasse mit jeweils einem Attribut von jedem Datentyp) mit 348,3 Millisekunden eine ähnlich lange Zeit benötigt, wie die Klasse mit einem Element.

Auch das Serialisieren und Deserialisieren von Klassenattributen und Rekursion wird anscheinend sehr gut optimiert, da sich im direkten Vergleich zu der einfachen Klassen keine wesentlichen zeitlichen Unterschiede ergeben.

Hier kommt auch gut zur Geltung, wie stark Java optimieren kann, denn bei der kleinen Klasse wird für jedes Element im Schnitt eine Millisekunde benötigt, was im krassen Gegensatz zu 0,000071 Millisekunden steht. Diese extreme Beschleunigung wird zum einen durch Caching-Mechanismen, zum anderen auch vom JIT-Compiler und der daraus folgenden Parallelisierung hervorgerufen.

Welche Mechanismen wo genau ansetzten und wie sie im speziellen angewendet werden, wurde im Rahmen dieser Arbeit nicht untersucht.

Bei langen Strings dauert das Deserialisieren plötzlich länger als das Serialisieren, was daher rührt, dass bei der Klassennamensuche der komplette String durchlaufen werden muss.

## 7.7 Zusammenfassung der Testergebnisse

Obwohl bei der Deserialisierung der ankommende String erst auf die zugehörige Klasse geprüft werden muss, ist die Deserialisierung deutlich schneller erledigt als eine Serialisierung. Dies lässt darauf schließen, dass es einfacher und schneller geht, Objekte zu bauen als diese zu speichern.

Die Serialisierung mit Jackson ist zum einen schnell, zum anderen aber auch sehr Arbeitsspeicher intensiv. Beim Testen konnte beobachtet werden, wie beim Serialisieren der allokierte Arbeitsspeicher sprunghaft ansteigt. Es zeigte sich, dass mindestens vier mal so viel RAM benötigt wird, wie die zu serialisierende Datei groß ist.

# 8 Bewertung und Vergleich zwischen JAXB und Jackson

Im folgenden wird eine vergleichende Bewertung von Jackson und JAXB vorgenommen und eine Empfehlung abgegeben.

Die vorliegende Arbeite behandelte nur die Serialisierung von Java-Instanzen mit der Hilfe von Jackson in JSON-Dokumente. Jedoch wurde in einer anderen Arbeit die Serialisierung von Java-Instanzen in XML-Dokumente mittels JAXB untersucht auf die sich diese Arbeit immer wieder bezieht.[Wal14]

Eine Serialisierung in Jackson-Dokumente anstelle von XML-Dokumente bringt zum einen den Vorteil, dass JSON-Dokumente immer Java und JavaScript konform sind. Zum anderen ist JSON durch einen geringeren Metadaten-Overhead im Regelfall kleiner als ein entsprechendes XML-Dokument.

Nachteilig an Jackson ist, dass es nicht wie JAXB nativer Bestandteil der Java-Bibliothek ist. Deshalb werden bei der Verwendung von Jackson zusätzliche Java-Bibliotheken benötigt. Dieser Nachteil kann selbst durch das modulare System von Jackson nicht aufgehoben werden, denn durch die Auslangerung vom „Jackson-Core“ aus anderen Jackson-Bestandteilen sind immer mindesten zwei Module nötig.

Beide Ansätze Jackson und JAXB benötigen für die Steuerung der Serialisierer Annotations. Dieser Ansatz ist weit verbreitet und findet auch oft bei anderen Bibliotheken Anwendung, wie zum Beispiel „JPA“ einer Datenbankbibliothek. Der Vorteil von Jackson ist, dass es nicht nur in der Lage ist, eigene Annotations zu verstehen, sondern mit zusätzlichen Bibliotheken können auch andere Annotations verstanden und verwendet werden.

JAXB hingegen hat keine Möglichkeit auf andere Annotationen zu reagieren und somit müssen bei der Verwendung von mehreren verschiedenen Serialisierern entweder JAXB-Annotationen genutzt werden oder es müssen verschiedene Annotations Verwendung finden.

Im direkten Vergleich ist Jackson etwas langsamer als JAXB, jedoch befinden sich die Unterschiede hier im Millisekundenbereich. Diese könnten jedoch beim Übertragen durch die geringere Größe von JSON aufgehoben werden.

Ein weiterer Vorteil ist auch, dass Jackson besser mit großen Dateien umgehen kann und spätestens hier Geschwindigkeitsvorteile bietet.

In Abwägung aller Vor- und Nachteile wird eine Verwendung von Jackson gegenüber JAXB vorgezogen, was die aufgezeigten Vor- und Nachteile belegen.

# 9 Zusammenfassung

Ziel der vorliegenden Arbeit war die Untersuchung der Möglichkeiten einer Serialiserung von Java-Instanzen in JSON-Dokumente und zurück. Nach der Untersuchung verschiedener Ansätze wurde in Zusammenarbeit mit einer anderen Arbeit von Herrn Achim Walz für die Verwendung Jackson entschieden. Herr Walz sollte es Serialisierung nach XML untersuchen und entschied sich in Zusammenarbeit für JAXB. [Wal14]

Nach einer Untersuchung der Möglichkeiten Instanzen in JSON zu speichern, wurde sich auf die Analyse und die Umsetzung mit Jackson spezialisiert. Es wurde untersucht welche Datentypen JSON nativ speichern kann und welche möglichkeiten die einzelnen Module der Jackson Bibliothek bieten.

Es wurde gezeigt das eine Vollständige Serialisierung von Java-Instanzen möglich ist,woraufhin auch untersucht wurde welche Möglichkeiten durch Annotationen möglich sind. Hier wurde auf die Arbeit von Herrn Walz verwiesen, da sich im Projekt dafür entschieden wurde JAXB-Annotationen zu verwenden.

Im laufe der Arbeit wurde ersichtlich das eine Spezifikation des SMD-Assistenten notwendig wird. Da die Serialisierer auf Daten vom Assistenten zugreift, wurde eine solche erste Spezifikation erstellt und ist nun in der vorliegenden Arbeit zu finden.

Ähnlich wie mit dem SMD-Assistenten wurde mit dem ClassGenerator verfahren. Da der CG die Klassen liefert welche die Serialisierer für ihre Arbeit benötigen wurde auch eine erste Spezifikation vom ClassGenerator erstellt.

Nach der Implementierung der Serialisierer, zusammen mit Herrn Walz, wurden Klassen erstellt, welche zu Testzwecken serialisiert wurden. Hierbei wurden alle Java eigenen „Standard-Datentypen“ getestet. Aber auch das Verhalten bei rekursiven Aufrufen sowie das einbinden von Klassen-Attributen wurde untersucht.

Beim serialisieren einer Klasse, welche Massendaten enthält, stellte sich heraus, das sowohl Jackson als auch JAXB Probleme haben diese Klasse zu serialisieren. Dies ergab sich durch einen Array-Überlauf in der Klasse `StringBuffer` welche von beiden intern Aufgerufen wird.

Bei der Bearbeitung der Aufgabenstellung wurden Konflikte mit dem OPM-Modell festgestellt, worauf einige Änderungen im OPM-Modell vorgeschlagen wurden, welche zum jetzigen Zeitpunkt zur Diskussion stehen.

Es wurde festgestellt, das eine Serialiserung mit Jackson Vorteile gegenüber JAXB bietet und eine entsprechende Empfehlung gegeben.

# 10 Ausblick

Im weiteren Verlauf des Projektes, wird nun die Spezifizierung des SMD-Assistent und des Class-Generators noch einmal überarbeitet und weiter auf dem Gebiet geforscht.

Der UDDE, welcher schon in einer ersten Version existiert, soll nun so erweitert werden das er nicht nur einfache Standard-Datentypen verstehen kann, sondern auch Klassen-Attribute sollen später durch ihn Verarbeitet werden können.

Wenn alle Bereiche und Bestandteile des GDS spezifiziert sind, wird eine erste Version nach OPM-Regeln implementiert.

# 11 Abkürzungsverzeichnis

**KIT** *Karlsruher Instituts für Technologie*

**GDS** *Generic Data Services*

**OPM** *Objektorientierten Programmiermodell*

**SMD** *Strukturelle Metadaten*

**JSON** *JavaScript Object Notation*

**IAI** *Institut für Angewandte Informatik*

**JAXB** *Java Architecture for XML Binding*

**UDDE** *User Data Description Editor*

**AMD** *Anwendermetadaten*

**CG** *Class Generator*

**IG** *Interface Generator*

# Literatur


[Fle] *FLEXJSON*. http://flexjson.sourceforge.net/

[Fle12] *Nie mehr ohne JSON Serializer*. http://blog-it.hypoport.de/2012/05/29/keine-debugging-session-ohne-json-serializer/. Version: Mai 2012

[Gri02] Griffiths, Alan: *More Exceptional Java*. http://accu.org/index.php/journals/399. Version: Juni 2002

[Jac14] *Jackson Project Home*. https://github.com/FasterXML/jackson. Version: August 2014

[Jah10] Jahn, Prof. Dr. Karl-Udo: *Theoretische Grundlagen der Informatik*. http://www.imn.htwk-leipzig.de/~jahn/Grundl09/gfol09.pdf. Version: Wintersemester 2009/2010

[JSO14] *Jackson JSON Schema Module*. https://github.com/FasterXML/jackson-module-jsonSchema. Version: August 2014

[Mky11] *Jackson Tree Model Example*. http://www.mkyong.com/java/jackson-tree-model-example/. Version: August 2011

[Sri13] Sripapasa, Sai S.: *JavaScript and JSON Essentials*. 2013

[Ste13] Sternberg, Ralf: *A Fast and Minimal JSON Parser for Java*. April 2013

[Wal14] Walz, Achim: *Serialisierung von Datenobjekten in XML zur Übertragung von Objektdaten aus Energieanwendungen*. 2014

[Wik14a] *JavaScript Object Notation*. http://de.wikipedia.org/wiki/JSON. Version: Juni 2014

[Wik14b] *Serialisierung*. http://de.wikipedia.org/wiki/Serialisieren. Version: Mai 2014

[Zil14] Zilian, Lars: *Strukturelle Metadaten- Objektorientierte Beschreibung von Klassen für das generische Management von Energiedaten und -Modellen*. 2014