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 von Datenobjekten in JSON zur Übertragung von Objektdaten.

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. Das GDS wird zur Zeit im *Institut für Angewandte Informatik* (IAI) des *Karlsruher Instituts für Technologie* (KIT) entwickelt und ist ein System generischer Datenservices welches bei Vertrigstellung 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.

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 auserhalb der Klasse nicht änderbar, und werden gegebenenfalls durch Getter- und Setter-Methoden aufgerufen
- Programme bestehen nur aus Objekten, der Aufruf erfolgt ausschließlich Über Methodenaufrufe
- Es können Standarddatentypen 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 muss den Spezifikationen der verwendete Sprache entsprechen

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

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 dem Anwendungsprogrammen zur Verfügung.

Die Metadaten werden in einer SQL-Datenbank gespeichert, aus der sie, wenn benötigt, geladen werden können. [Zil14]

2 Aufgabenstellung

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

Für den Transfer von solchen Anwendungsdaten sind Schnittstellen für die automatische Serialisierung der 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. Die Empfängerseite muss letztendlich in der Lage sein aus den übertragenen Daten wieder Objekte zu erstellen. Da es sich im Projekt bei allen Klassen handelt, muss es prinzipiell möglich sein jedes Objekt zu serialisieren.

Abschließend soll ein Vergleich zeigen welche der Serialisierungsarten (JSON oder XML) 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

JavaScript Object Notation (JSON) ist ein textbasiertes Format zum Datenaustausch, wobei jedes gültige JSON-Dokument auch ein gültiges JavaScript ist. Jedoch ist JSON unabhängig von der Programmiersprache. Es wurde als Ersatz für XML geschaffen und wird hauptsächlich in Bereichen eingesetzt wo Ressourcen wie Speicherplatz, Prozessorleistung und Netzwerkverbindung stark limitiert sind. Im Aufbau erinnert JSON 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 unten 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]

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.

Zeile zwei enthält einen String, eine Zeichenkette in der jedes Zeichen erlaubt ist. Der boolsche Wert wird genau wie ein Nullwert ohne Anführungszeichen geschrieben wie in den Zeilen 3 und 4 gezeigt.

Zahlen können ganzzahlig, Fließkommazahlen oder Exponentialzahlen sein.

Ein Array kann mehrere Werte enthalten und wird deshalb von eckigen Klammern umschlossen. Die eigentlichen Werte im Array müssen jedoch vom selben Typ sein.

Objekte können wiederum Objekte enthalten, wie es in den Zeilen acht bis zehn dargestellt ist. Das innere Objekt wird wieder von geschweiften Klammern umschlossen.

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

Ein Beipiel für eine zu serialisierende Klasse, ohne Annotationen für einen Serialisierer, könnte wie folgt aussehen. Worum es sich bei Annotationen genau handelt wird im Kapitel 5 genauer beschrieben.

```
1 public class JsonObjects {
2     public String string = "Variable123";
3     public boolean bool = true;
4     public String nullwert = null;
5     public int zahl = 1234567;
6     public double fliesskomma = 1230000.0;
7     public String array[] = new String[]{"Beispiel", "fuer", "Array"};
8     public OtherObject object = new OtherObject();
9 }
```

Das selbe Beispiel als serilalisiertes JSON-Objekt würde dann wie folgt aussehen können. Je nachdem, mit welcher Biliotheke Serialisiert wird, beziehungsweise welche Annotationen an den Quellcode noch zusätzlich angebracht sind, kann das JSON-Objekt auch anders aussehen.

```
{
    "stringValue": "Variable123",
    "boolValue": true,
    "nullValue": null,
    "zahlValue": 1234567,
    "fliesskommavalue": 1.23e+6,
    "arrayValue": ["Beispiel", "fuer","Array"],
    "objekt": {
        "zahl_2Value": "1234",
    }
}
```

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, das für eine grundlegende Verwendung von JSON keine zusätzlichen Bibliotheken benötigt werden.

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

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, unter der Bedingung, dass das JSON-Objekt als Variable mit dem Namen `JSONVariablenName` im JavaScript deklariert wurde, zurück.

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.

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.

4.2 Möglichkeiten der JSON-Serialisierung in Java

Grundsätzlich gibt es verschiedene Möglichkeiten eine JSON-Serialisierung in Java durchzuführen. Im folgenden werden die im Projektteam diskutierten Möglichkeiten genauer vorgestellt.

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 bezeichnit wird.

Dieser Scanner muss in der Lage sein einen JSON-Datenstrom zu lesen und ihn in die einzelnen Bestandteile aufspalten.

Eine weitere Funktion die erfüllt werden muss, ist die eines Parsers. Dieser muss die einzelnen vom Scanner erkannten Bestandteile in Javaobjekte umwandeln.

Bei der Implementierung muss des weiteren zum Beispiel auf Rekursion und nicht valide JSON-Objekte geachtet werden.

Es muss also darauf gehend geprüft werden, ob irgendwo eine Rekursion vorliegt und wie diese behandelt werden soll. Dies bezieht sich nicht nur auf die lesende, sondern auch auf die schreibende Richtung.

4.2.2 Flexjson

Flexjson ist eine einfache Bibliothek für das Serialisieren und Deserialisieren von JSON-Objekten in Javaobjekte.

Wenn Attributnamen in JSON von dem Deklarationsnamen im Javaobjekt abweichen sollen, müssen Annotationen verwendet werden.

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

4.2.3 Jackson

Jackson ist ähnlich wie Flexjson eine Bibliothek für die Serialisierung von Javaobjekten zu JSON-Objekten. Vorteilhaft an Jackson ist, dass die Bibliothek modular aufgebaut ist. So wird für eine einzelne Aufgaben nicht die gesamte Bibliothek benötigt. Um nur spezielle Aufgaben zu erfüllen, reicht es aus Teile der Jackson-Bibliothek einzubinden.

Ein weiterer Vorteil ist, dass es über ein Jackson-Modul möglich ist Annotationen von der *Java Architecture for XML Binding* (JAXB) zu verwenden. Das ermöglicht es, für die JAXB und Jackson Serialisierung die selben Annotationen zu nutzen. Dies würde die Bearbeitung und die Übersichtlichkeit sehr vereinfachen, da nicht zwei unterschiedlichen Annotationen 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 ist und keine zusätzlichen „technischen Schulden“ gemacht werden.

Zum anderen ist es sehr Aufwändig eigen Klassen für die Serialisierung und Deserialisierung zu schreiben, da einfach zu viele Sachen beachtet und berücksichtig werden müssen. Dies ist im Rahmen dieser Arbeit leider nicht möglich.

Aus diesem Grund schied 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.

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

Der große Vorteil der Steuerung über Annotationen ist jedoch auch ein Nachteil, den diese müssen in allen Klassen angebracht werden in denen an den Attributen etwas zu beachten ist, wie das sie nicht serialisiert werden sollen.

Dies war am Schluss auch das entscheidende Element warum sich für eine Umsetzung mit Jackson und JAXB entschieden wurde.

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 nach der Besprechung

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

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.

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 drei Hauptmodulen, welche wie folgt bezeichnet sind:

- „jackson-core“ welches die JSON spezifische Implementierung sowie eine low-level streaming API enthält
- „jackson-annotations“ welches die Jackson spezifischen Annotationen enthält.
- „jackson-databind“ welches für das *databind* verantwortlich ist.
- „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.

Die Jackson Annotationen enthalten Informationen die für das Serialisieren beziehungsweise Deserialisieren verantwortlich sind. Jackson kann aber auch Annotationen von anderen Serialisierern erkennen und darauf reagieren.

Eine Mischung von Annotationen aus verschiedenen Serialisierern ist Grundsätzlich möglich, aber nicht empfehlenswert da hier die Übersichtlichkeit und die Verständlichkeit des Codes leidet.

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

Mit diesen drei Modulen ist Jackson 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 Annotationen für Jackson und JAXB zu haben, wird ein weiteres Jackson-Modul benötigt, welches in der Lage ist die JAXB-Annotationen zu verarbeiten.[Jac14]

5.2 Serialisierung mit Jackson

Der im folgenden beschriebene Sachverhalt erläutert das Quellcode-Listing unten.

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

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.

Dem `inspector` wird eine `TypeFactory` mit „Default-Einstellungen“ übergeben wird. Dies bedeutet es wird auf die Original JAXB-Annotationen geparst, ohne auf Sonderfälle zu achten. Andere Annotation werden nicht berücksichtigt. Der `inspector` wird nun dem `mapper` übergeben, damit dieser auf die entsprechnden Annotationen 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 puren I/O-Probleme sind. Die Exception erbt von `IOExecption`.

```
1  public String serialize(OPMObject opmObject) {
2      ObjectMapper mapper = new ObjectMapper();
3      AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(
           TypeFactory.defaultInstance());
4      mapper.setAnnotationIntrospector(inspector);
5
6      if (opmObject == null) {
7              throw new IllegalArgumentException("OPMObject␣can␣not␣be␣null!");
8              }
9      try {
10         return mapper.writeValueAsString(opmObject);
11     } catch (JsonProcessingException e) {
12         e.printStackTrace();
13         return null;
14     }
15 }
```

5.3 Deserialisierung mit Jackson

Der im folgenden beschriebene Zusammenhang ist noch einmal im Quellcode-Listing unten zu finden.

Für die Deserialisierung mit Hilfe von Jackson wird wie bei der Serialisierung ebenfalls ein Databinder und AnnotationInspector benötigt, welche wie im Kapitel 5.2 erstellt werden.

Bevor dies jedoch passiert, wird geprüft ob der eingegebene String weder `null` noch `empty` ist. Sollte das der Fall sein, wird die Methode `readValue` vom `mapper` mit dem übergebenden String und der Information um welche Klassen-Instanz (`.class`-File)es sich beim String handelt übergeben. Die zurückgegebene Instanz entspricht dem Typ der übergebenen Instanz.

Die Schwierigkeit beim Deserialisieren besteht also 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, das es sich immer um eine Instanz der Klasse „TestData“ handelt. Wie diese Einschänkung aufgehoben werden kann wird im folgenden Kapitel beschrieben.

```
1  public <T extends OPMObject> T deserialize(String string) {
2          if (string == null) {
3              throw new IllegalArgumentException("String can not be null!");
4          }
5          if (string.isEmpty()) {
6              throw new IllegalArgumentException("String can not be empty!");
7          }
8
9          ObjectMapper mapper = new ObjectMapper();
10         AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(
               TypeFactory.defaultInstance());
11         mapper.setAnnotationIntrospector(inspector);
12         try {
13             return (T) (mapper.readValue(string, Class.forName("opm_serializer.
                   TestData")));
14         } catch (IOException | ClassNotFoundException | ClassCastException e) {
15             e.printStackTrace();
16         }
17         return null;
18 }
```

5.4 Instanzunabhängige Deserialisierung

Wie gerade schon erkläutert, wird beim Deserialisieren vorausgesetz, dass die Instanz des zu deserialisierenden Strings bekannt ist.

Um den String nun eindeutig einer Instanz zuzuordnen musste eine eindeutige Kennzeichnung geschaffen werden.

Es wurde sich darauf geeinigt, das ein String der Klassenname über ein Attribut `className` hinzugefügt wird. Aus diesem Grund bekam die Klasse `OPMObject` eine String-Attribut `className` in welchem der Klassenname der jeweiligen Klasse abgelegt ist.

Daraus ergibt sich, das 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 String geschrieben und dieser kann dadurch eindeutig identifiziert werden.

Um nun den Klassennamen aus den String zu lesen, wurde die neue Methode `getClassFileFromString` geschrieben, welche den Klassennamen aus dem String filtert und diesen dann zurückliefert. Die wurde mit File der `split`-Methode der String-Klasse realisiert.

5.5 Klassendiagramm der Serialisierung

Wie von OPM verlangt erben hier alle Klassen von `OPMObject`. Um diese Arbeit und auch die Geschwindigkeit, beziehungsweise die Funktionalität der XML- und JSON-Serialisierung vergleichen zu können wurde sich mit Herrn Achim Walz auf eine gemeinsame abstrakte Klasse `Serializer` geeinigt.

Die Klasse `JSONSerializer` erbt um einen direkten Vergleichen durchführen zu können genau wie `XMLSerializer` von `Serializer`. `Testdata` und `TestData2OPM` sind erste Test-Klassen von denen Instanzen serialisiert und deserialisiert werden.

Main-Klasse in diesem Projekt ist `OPM_Serializer`. Die `main`-Methode setzt die Serialisierung im Test in Gang.

Damit wie gewünscht jede Klasse serialisiert werden kann, wurde die Methode `serializeMe` zu OPMObject hinzugefügt. Diese Methode nutzt nun bei Aufruf die `serialize`-Methode des jeweiligen Serialisierers.

Ein vollständiger Überblick ist im Klassendiagramm unten zu finden gegeben.

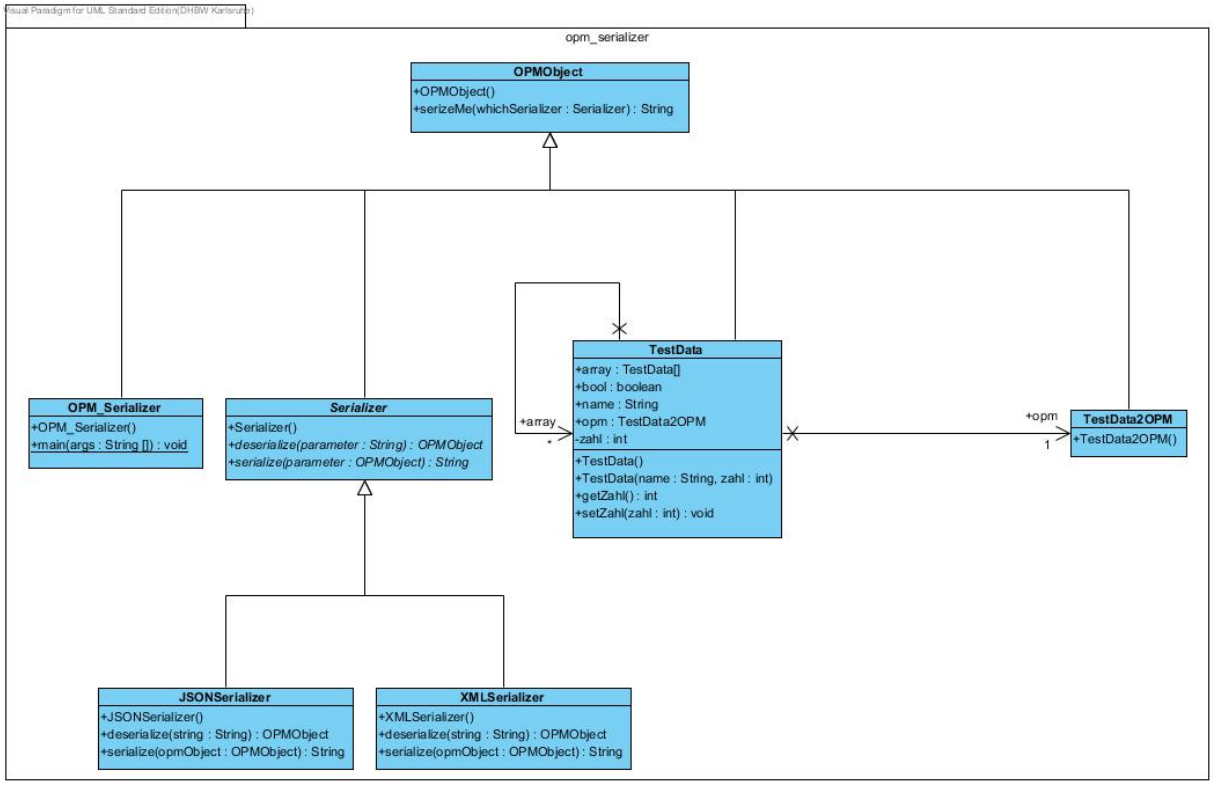

Abbildung 1: Klassendiagramm der Serialisierung

5.6 JSON-Schema Erstellung aus einer Klasse

Ähnlich wie auch bei XML gibt es in JSON ein Datenformat, welches die zu erwartende Form des Streams vorgibt. Wie auch in 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.

Somit steht am Anfang eines JSON-Schemas immer der Bezeichner `type` mit dem entsprechenden Wert, nämlich `object`. 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 Wert des Attributs weiter eingränzen. Sie werden jedoch erst angezeigt, wenn die entsprechenden Annotationen im Quellcode angebracht sind. Auf weiter Bezeichner wird im folgenden jedoch nicht weiter eingegangen, da dieses Thema zu umfangreich ist um es in dieser Arbeit weiter zu vertiefen.

Im Beispiel wird einfach nur der Typ des Attributs bezeichnet, jedoch ist auch möglich weitere Einschränkungen über Annotationen zu machen. Mit dessen Hilfe könnte zum Beispiel ein Wertebereich für Zahlen angeben werden.

Am Anfang einer Propertie steht der Name des Atrributs wie im Quelltext (im Beispiel `"stringValue"`), wie im Beispiel unten. 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.

Die Einschränkungen und Beschreibungen der Attribute werden dem Serializer mit Hilfe der Annotationen übergeben, welche Java-Methodennamen enthalen . 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 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 folgenden 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 dirket behandelt wird. Des weiteren wird für die Schemaerzeugung noch ein `SchemaFactoryWrapper` erstellt, welcher das Schema erstellt. Der Schema-Wrapper kann unter umständen die JsonProcessingException werfen.

Dem Wrapper wird nun über den nächsten Methodenaufruf auf den Mapper und die entsprechend zu Mappende Instanz angesetzt. Dies geschieht über die Methode `acceptJsonFormatVisitor`.

Mittels der Methode `finalSchema` wird das Schema der Instanz 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 Attribute 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.

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.

Um eine Serialisierung von allen Klassen zu gewärleisten muss eventuell das OPM-Modell angepasst werden. Da alle Klassen sich an die OPM-Regeln halten, ist somit gewährleistet das alle ankommenden Instanzen serialisiert oder deserialisiert werden können.

In ersten Tests mit dem JSONSerializer wurde die Funktionsfähigkeit bewiesen. Jedoch ist der derzeitige Stand des Serialisierers noch nicht in der Lage verschiedene Klassen zu deserialisieren. Bisher ist der Klassenname fest vorgebeben.

Im Projekt aber sollen unterschiedlichste Klassen deserialisiert werden können. Das wirft die Frage auf wie Festgestellt wird, um welche Klasse es sich bei der Instanz handelt. Dieser Punkt wird nun in den folgenden Kapiteln ausfühlich erläutert.

5.8.1 Auffälligkeiten beim erstellen des JSON-Schemas

Nach einigen Tests wurde festgestellt, das Jackson für alle Zahlen immer `integer` im Schema angibt, obwohl JSON auch `double` oder `float` kennt. Egal ob es sich in der Klasse um `float`, `double` oder `integer` handelt. Warum dies im Schema nicht Ordnungsgemäß übernommen wird, konnte nach einiger recherche nicht herausgefunden werden.

6 Der SMD-Assistent im GDS-System

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

In einer MySQL-Datenbank können alle Klasseninformationen abgelegt werden. Zu diesen Informationen zählen zum Beispiel der Klassenname, Attribute mit Modifier, Methoden mit Modifier Parametern und Rückgabetyp.

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 *Anwender Metadaten* (AMD) in das System. Unter AMD werden die zu seinen Klassen passenden Instanzen bezeichnet, welche der User der Anwendung zum Speichern übergibt. Des weiteren legt der UDDE die generierten „strukturellen Metadaten“ in einer Datenbank ab, welche später vom SMD-Assistent ausgelesen werden können.

Der SMD-Assistent ist also für das Umwandeln von Klassen in SMDs verantwortlich. Nach dem erstellen der SMDs werden diese zur Aufbewahrung vom Assistenten dem GDS übergeben, welcher die Ablage der Daten in einer Datenbank verwaltet.

Die SMDs werden von der Anwendung an verschiedenen Stellen benötigt. Der *Class Generator* (CG) wandelt die SMDs wieder in Java-Klassen, welche OPM-Konform erzeugt werden. Eine genaue Spezifikation für den Klassengenerator ist im Kapitel 6.2 zu finden. Die generierten Datenströme könnwn dann ebenfalls vom GDS in einer Datenbank gespeichert werden.

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

Der beschriebene Zusammenhang der Komponenten des GDS ist im Bild unten noch einmal verdeutlicht.

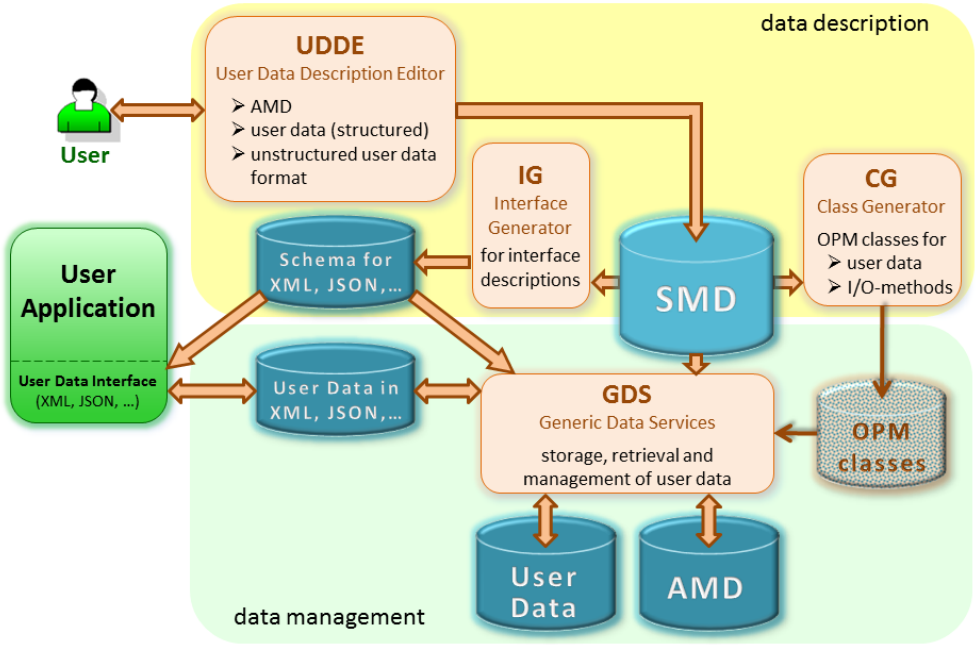

Abbildung 2: GDS Übersicht

6.1 Spezifikation des SMD-Assistent

Im Verlauf der Arbeit wurde zunehmend klar, das die Spezifikation des SMD-Assistent nötig wird. Da er, wie im Schaubild oben zu erkennen, zentraler Bestandteil des Projektes ist. Der SMD-Assistent wurde im Projekt öffter diskutiert und soll an dieser stelle einmal genauer erleutert werden.

6.1.1 Funktionen des SMD-Assistent

Der SMD-Assistent soll die „strukturellen Metadaten“ aus der Datenbank laden und diese an das GDS, den IG oder den CG weiterreichen.

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

Eine Instanz der Klasse `ClassDecr` enthält mittels der Klassen `MethodDescr` und `AttrDecr` alle nötigen Informationen um eine Klasse rekonstruieren zu können.

`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 Type und Modifier. [Zil14]

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

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

Damit es nicht zu einem Arbeitsspeicher überlauf kommt, muss der SMD-Assistent genau wie alle anderen Assistenten auch eine Möglichkeit besitzen den Zwischenspeicher 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.

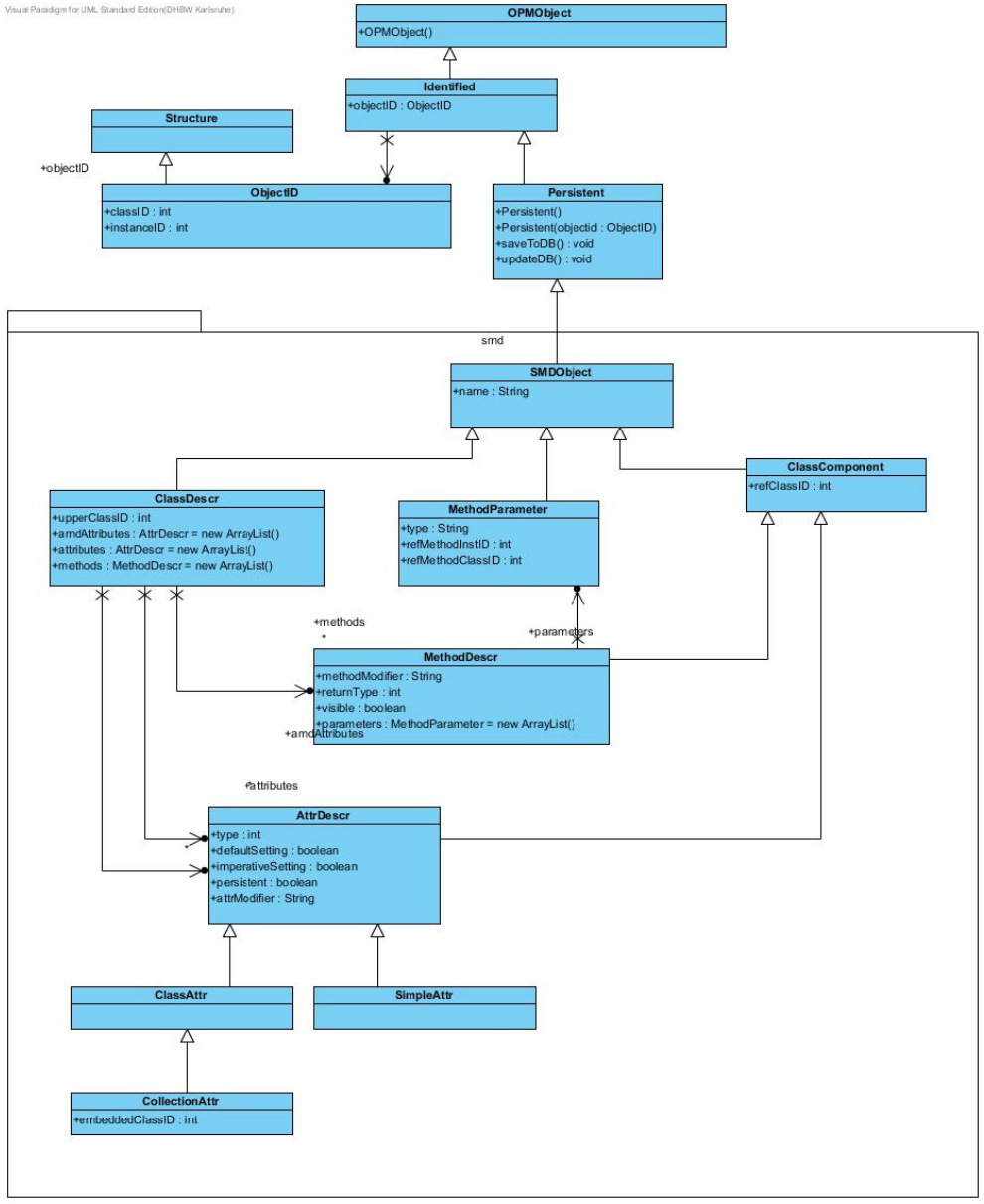

Abbildung 3: Klassendiagramm der SMD-Klassen

6.1.2 Der SMD-Assistent unter Java

Unter Java kann die Arbeitsweise des SMD-Assistent deutlich vereinfacht werden, da hier die SMDs nur bei explizitem verlangen geliefert werden müssen, denn Java verwendet mit den *.class*-Files eigene Metadaten über die eine Verarbeitung unter Java einfacher Umgesetzt werden kann. Der SMD-Assistent sollte unter Java also in der Lage sein direkt *.class*-Files zu liefern.

6.2 Der Klassengenerator

Der Klassengenerator ist ein Modul des GDS, welcher aus den vom SMD-Assistenten gegebenen strukturellen Metadaten wieder Klassen generiert. Es gibt eine abstrakte Klasse `ClassGenerator` welche abstrakte Methoden zur Verfügung stellt.

Für jede Programmiersprache, soll nun ein Klassengenerator von der abstrakten Oberklasse abgeleitet werden. Die Programmiersprachen spezifischen Klassengeneratoren sind dann in der Lage, für ihre Programmiersprache aus den SMDs 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 SMDs auftauchen. Des weiteren muss er diese Methoden auch nach 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 Annotationen an den Klassen Notwendig, welche ebenfalls vom `ClassGenerator` angebracht werden müssen.

7 Test des JSON Serialisierers

Im Verlauf dieses Kapitels wird die Arbeitszeit und das Verhalten der Jackson Bibliotheken 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 Bilbliotheken genauer zu beurteilen.

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

7.2 Testaufbau der 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 Daten weiter Serialisiert werden und somit sollte die Bearbeitung des 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 Aunlaufen der Serialisierer vernachtlässigt 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 dritte Testklasse entält von jedem möglichen „Primitiven Datentypen“ genau ein Attribut mit einem Wert. Sie dient Hauptsächlich als Vergleichspunkt zu den beiden noch fehlenden Klassen, aber auch um zu beweisen das jeder Datentyp Serialisiert werden kann.

Eine weitere Klasse ist genau wie die dritte beschriebene Vergleichsklasse aufgebaut, jedoch mit dem Unterschied das diese eine weiteres Attribut besitzt. Bei dem neuen Attribut, handelt es sich um ein Klassenattribut, welches der dritten Klasse entspricht. Geprüft soll nun werden ob die Serialisierung ähnlich lang wie der dritte Klasse dauert. Sollte das der Fall sein beinhalten die eingesetzten Serialisierer keinen Cache, bzw. arbeiten nicht 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 die Serialisierung mit Rekursion umgeht.

7.3 Testaufbau der 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 2,33GHz
- RAM: 8GB DDR-2 667MHz (Zugriffszeit 1,5ns)
- Chipsatz: Intel 5000 Series Chipset
- Festplatte: Seagate ST31000528AS (1TB) davon 16GB Swap

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

7.4 Testdurchführung

Die Testdurchführung wurde zuerst Zusammenhängend ausgeführt, jedoch wurden sehr früh Effekte verschiedener Caching-Mechanismen festgestellt, was eine genaue Messung der Tests nicht möglich macht.

Es wurde somit entschieden jeden Test einzeln durchzufühen und ihn jeweils drei mal zu wiederholen, wenn keine extremen Schwankungen auftreten. Sollte des der Fall sein muss ein Test öfters wiederholt werden.

7.4.1 Auffälligkeiten bei Testbeginn

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

Dies liegt daran, dass bei einer solch großen Datenmenge extrem Große Strings entstehen welche die `StringBuffer`-Klasse 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 das sich nur noch 134217728 `short`-Werte im Array befinden.

7.5 Testergebnisse

Die folgenden Testergebnisse wurde wie oben beschrieben ermittelt.

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 alle 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.

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

Der selbe 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

Über JSON kann also Ausgesagt werden, dass das Starten des Serialisierungsprozesses in etwa 336 Millisekunden dauert, da das Serialisieren einer Klasse mit einem Element im Mittel diese Zeit benötigt und somit die Serialisierung dieses Attributs vernachlässigt werden kann.

Dies ist nur möglich, da auch eine Klasse mit mehreren Elementen (Klasse mit jeweils ein Attribut von jedem Datentyp) mit 348,3 Millisekunden eine ähnlich Zeit benötigt.

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

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

Welche Mechanismen wo genau aunsetzten und wie sie im speziellen Angewendet werden, wurde im Rahmen dieser Arbeit nicht untersucht

Auch das Serialisieren und Deserialisieren von Inneren Klassen und Rekursion wird anscheinend sehr gut Optimiert, da sich hier im direkten Vergleich zu einfachen Klassen keine wesentlichen Zeitlichen Unterschiede ergeben.

7.7 Zusammenfassung der Testergebnisse

Obwohl bei der Deserialisierung der ankommende String erst auf die Zugehörige Klasse geprüft werden muss ist des 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, aber auch sehr Arbeitsspeicher intensiv. Beim Testen konnte beobachtet werden wie beim Serialisieren der allokierte Arbeitsspeicher sprunghaft ansteigt. Es zeigte sich, das mindestens vier mal so viel Ram benötigt wird, wie die Datei eigentlich 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 das Kapitel immer wieder bezieht.[Wal14]

Eine Serialisierung in Jackson-Dokumente anstelle von XML-Dokumente bringt zum einen den Vorteil das 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-Bibliotheke ist. Des 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“ sind immer mindesten zwei Elemente nötig. Zum einen der Core und die aufgabenspezifischen Bibliotheke.

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

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

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

Ein weiterer Vorteil ist auch das Jackson besser mit großen Dateien umgehenkann 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 Bibliotheke 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 für JAXB-Annotationen entschieden wurde..

Im laufe der Arbeit wurde ersichtlich das eine Spezifikation des SMD-Assistenten notwendig wird. Da die Serialisierer auf Daten vom Assistenten zugreift, wurde eine erste Spezifikation erstellt.

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

Nach der Implementierung der Serialisierer wurden Testklassen erstellt, welche serialisiert wurden. Hierbei wurden alle Java eigenen „primitiven Datentypen“ getestet. Aber auch das Verhalten bei rekursiven und 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 Aufgabe wurden aber auch einige Änderungen im OPM-Modell vorgeschlagen, 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 ClassGenerators 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 Datentypen verstehen kann, sondern auch Klassen-Attribute sollen später durch ihn Verarbeitet werden.

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 *Anwender Metadaten*

CG *Class Generator*

IG *Interface Generator*

Literatur

[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

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

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