

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	5
1.1	Generisches Managementsystem für Energiedaten	5
1.2	Objektorientiertes Datenmodell	5
1.3	Strukturelle Metadaten	5
2	Aufgabenstellung	6
3	JavaScript Object Notation	7
3.1	Der Aufbau von JSON	7
3.2	JSON in Verbindung mit Programmiersprachen	7
3.2.1	JSON und JavaScript	7
4	Objektserialisierung nach JSON	9
4.1	Was ist Serialisierung	9
4.2	Möglichkeiten der JSON-Serialisierung in Java	9
4.3	Eigener Ansatz	9
4.4	Flexjson	9
4.5	Jackson	9
4.6	Auswertung der Möglichkeiten	10
4.7	Fragestellungen nach der Besprechung	10
5	Jackson und das Jackson Projekt	11
5.1	Jackson-Module	11
5.2	Serialisierung mit Jackson	11
5.3	Deserialisierung mit Jackson	12
5.4	Klassendiagramm der Serialisierung	12
5.5	JSON-Schema Erstellung aus einer Klasse	13
5.6	JSON-Schema mit Hilfe von Jackson erstellen	14
5.7	Auffälligkeiten beim Testen	15
5.7.1	Auffälligkeiten beim erstellen des JSON-Schemas	15
6	Der SMD-Assistent im GDS-System	16
6.1	Spezifikation des SMD-Assistent	17
6.1.1	Funktionen des SMD-Assistent	17
6.1.2	Der SMD-Assistent unter Java	18

7	Test des JSON Serialisierers	19
7.1	Testaufbau	19
7.2	Testaufbau der Software	19
7.3	Testaufbau der Hardware	20
7.4	Testdurchführung	20
7.4.1	Auffälligkeiten bei Testbeginn	20
7.5	Testergebnisse	20
7.6	Testzusammenfassung und Auswertung	21
7.7	Zusammenfassung der Testergebnisse	21
8	Abkürzungsverzeichnis	22

1 Einleitung

Die folgende Arbeit befasst sich mit der Serialisierung von Datenobjekten in JSON zur Übertragung von Objektdaten.

1.1 Generisches Managementsystem für Energiedaten

Das *generisches Managementsystem für Energiedaten* (GDS) 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 Release vollautomatisch Energiedaten managen können 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 dargestellt:

- Basisklasse `OPMObject` von der alle Klassen Erben
- Es gibt keine Konstanten
- Attribute sind Grundsätzlich `private` und werden durch Getter- und Setter-Methoden aufgerufen
- Programme bestehen nur aus Objekten, der Aufruf erfolgt ausschließlich Über Methodenaufrufe
- Bezeichner werden im „Camel Case“ formuliert
- In der Dokumentation eines Attributs wird immer der erlaubte Wertebereich spezifiziert
- Die Dokumentation muss den Spezifikationen der verwendete Sprache entsprechen
- Jede Klasse enthält einen der Stati `Valid`, `Experimental` oder `Depricated` in ihrer Dokumentation

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, wo sie wenn benötigt geladen werden können. [Zil14]

2 Aufgabenstellung

In einem GDS-System werden Anwendungsdaten mit Hilfe von Objekten verwaltet, welche den Regeln von OPM folgen und mittels der SMDs 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.

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. 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 Beispiel für 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 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 boolesche Wert wird genau wie ein Nullwert ohne Anführungszeichen geschrieben wie in den Zeilen drei und vier gezeigt.

Zahlen können Ganzzahlig, Fließkommazahlen oder Exponentialzahlen sein wie sie auch im Beispiel zu finden sind.

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 beschrieben werden.

```
1 {  
2     "String": "Variable123",  
3     "Boolean": true;  
4     "Nullwert": null,  
5     "Zahl": 1234567  
6     "Fließkomma": 1.23e+6,  
7     "Array": ["Beispiel", "fuer", "Array"],  
8     "Objekt": {  
9         "Zahl_2": "1234",  
10    }  
11 }
```

3.2 JSON in Verbindung mit Programmiersprachen

Viele Programmiersprachen wie PHP, Python, C#, C++ und Java unterstützen JSON sehr gut und sogar Nativ.

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 Kommando `alert(JSONVariablenName.Zahl);`

liefert den Wert 1234567 aus dem Beispiel, unter der Bedingung das, dass JSON-Objekt als Variable `JSONVariablenName` 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 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 somit 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.

4.3 Eigener Ansatz

Eine Möglichkeit einen Funktionsfähigen Serialisierer zu erhalten, ist diesen selber zu schreiben. Hierfür müsse 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.

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.

4.4 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.

Nachteilig ist das beim Serialisieren immer explizit angegeben werden muss wenn geschachtelte Objekte mit serialisiert werden sollen.

4.5 Jackson

Jackson ist ähnlich wie Flexjson eine Bibliothek für die Serialisierung von Javaobjekten zu JSON-Objekten. Vorteilhaft an Jackson ist das die Bibliothek modular aufgebaut ist. So benötigt man für einzelne Aufgaben nicht die gesamte Bibliothek.

Ein weiterer Vorteil ist, dass es über ein Jackson-Modul möglich ist Annotationen von der *Java Architecture for XML Binding* (JAXB) zu verwenden. Was es ermöglicht 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.

4.6 Auswertung der Möglichkeiten

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

Vorteil des eigenen Ansatzes ist es zum einen, dass man hier völlig frei von fremden Bibliotheken ist und man keine zusätzlichen „technischen Schulden“ aufnehmen muss. Zum anderen ist es sehr Aufwändig eigen Klassen für die Serialisierung und Deserialisierung zu schreiben und 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 expliziert anzugeben welche Attribute serialisiert werden sollen, und welche nicht, wie es später einmal vorgesehen werden soll.

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.

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]

4.7 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 gegebenen Falls 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. 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.

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

Um eine Serialisierung mit Jackson umzusetzen wird zuerst eine Instanz der Klasse `ObjectMapper` benötigt, welche den Databinder darstellt. Der `mapper` ist somit für die Convertierung von Java-Instanzen zu JSON-Dokumenten verantwortlich.

Jedoch wird nicht nur der „Converter“ benötigt, sondern auch ein `AnnotationInspector`. Der `inspector` wird als Instanz von `JaxbAnnotationInspector` erstellt, welchem eine `TypeFactory` mit „Default-Einstellungen“ übergeben wird. Dies bedeutet es wird auf die Original JAXB-Annotationen geparkt, ohne auf Sonderfälle zu achten. Andere Annotation werden nicht berücksichtigt. Der `inspector` wird nun dem „Converter“ übergeben, damit dieser auf die entsprechenden Annotationen reagieren kann.

Um eine *Minimale Exception Safety* zu garantieren wird nun eine `Null`-Abfrage des zu serialisierenden Elements gemacht. Mit dieser Stufe der Sicherheit soll nicht verhindert werden das eine Exception passiert. Es wird lediglich garantiert das die Methode ohne Abzustürzen durchlaufen werden kann. [Gri02]

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

Dieser Zusammenhang ist noch einmal im folgenden Quellcode-Beispiel beschrieben.

```

1 ObjectMapper mapper = new ObjectMapper();
2 AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(TypeFactory.
    defaultInstance());
3 mapper.setAnnotationIntrospector(inspector);
4
5 if (opmObject == null) {
6     throw new IllegalArgumentException("OPMObject_cannot_be_null!");
7 }
8 try {
9     return mapper.writeValueAsString(opmObject);
10 } catch (JsonProcessingException e) {
11     e.printStackTrace();
12     return null;
13 }

```

5.3 Deserialisierung mit Jackson

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` mit dem übergebenden String und der Information um welche Klassen-Instanz es sich beim String handelt übergeben.

Die Schwierigkeit beim Deserialisieren besteht also nun darin, das 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 Einschränkung 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_cannot_be_null!");
4     }
5     if (string.isEmpty()) {
6         throw new IllegalArgumentException("String_cannot_be_empty!");
7     }
8
9     ObjectMapper mapper = new ObjectMapper();
10    AnnotationIntrospector inspector = new JaxbAnnotationIntrospector(
        TypeFactory.defaultInstance());
11    mapper.setAnnotationIntrospector(inspector);
12    // mapper.enableDefaultTyping();
13    try {
14        return (T) (mapper.readValue(string, Class.forName("opm_serializer.
            TestData")));
15    } catch (IOException | ClassNotFoundException | ClassCastException e) {
16        e.printStackTrace();
17    }
18    return null;
19 }

```

5.4 Klassendiagramm der Serialisierung

Wie von OPM verlangt erben hier alle Klassen von OPMObject. Um diese Arbeit mit der von Herrn Achim Walz vergleichen zu können wurde sich 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.

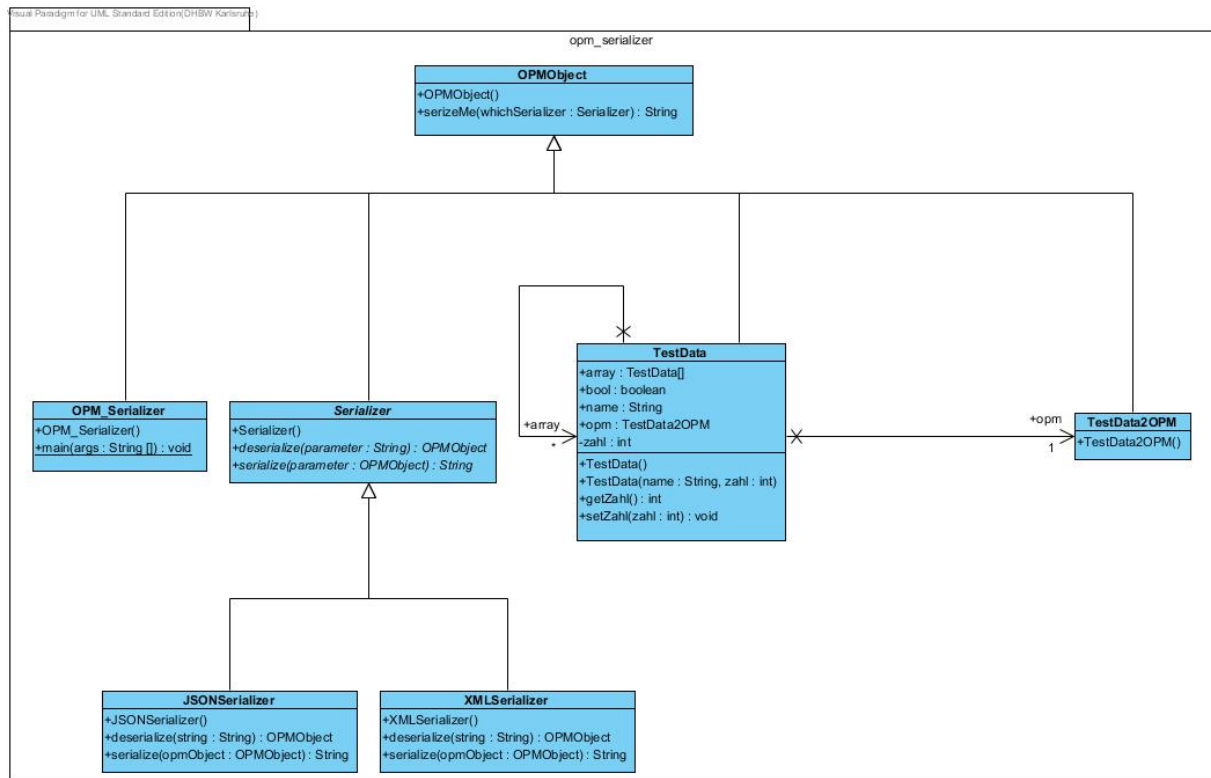


Abbildung 1: Klassendiagramm der Serialisierung

5.5 JSON-Schema Erstellung aus einer Klasse

Ähnlich bei 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.

Am Anfang einer Property steht der Attributname "`String`", wie im Beispiel unten. Gefolgt wird dieser Name von einem Objekt, welches die genauen Eigenschaften des Attributs beschreibt.

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

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-Methoden-Namen enthalten. Das setzen der Beschreibungen geschieht über Java-Methoden welche zur Laufzeit vom Serializer aufgerufen werden.

Auf die Erklärung der einzelnen Annotationen von JAXB wird an dieser Stelle verzichtet, da dieses Thema zu umfangreich ist um es in dieser Arbeit abzuhandeln.

```
1 {
2     "type": "object",
3     "properties": {
4         "String": {
5             "type": "string"
6         },
7         "Boolean": {
8             "type": "boolean"
9         },
10        "Nullwert": {
11            "type": "string"
12        },
13        "Zahl": {
14            "type": "integer"
15        },
16        "Fließkomma": {
17            "type": "integer"
18        },
19        "Array": {
20            "type": "array",
21            "items": {
22                "type": "string"
23            }
24        },
25        "Object": {
26            "type": "object",
27            "properties": {
28                "Zahl_2": "integer"
29            }
30        }
31    }
32 }
```

5.6 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.

Wie schon in früheren Beispielen gezeigt wird zu erst wieder ein `ObjectMapper` erstellt. Hierbei kann eine `JsonMappingException` auftreten, welche entweder weiter geworfen werden kann oder direkt 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.

Im Codebeispiel unten wird dieser Zusammenhang noch einmal verdeutlicht.

```
1 public String generateSchema(OPMObject opmObject) throws JsonProcessingException
2     , JsonMappingException{
3     ObjectMapper m = new ObjectMapper();
4     SchemaFactoryWrapper visitor = new SchemaFactoryWrapper();
5
6     m.acceptJsonFormatVisitor(m.constructType(opmObject.getClass()),
7         visitor);
8     JsonSchema jsonSchema = visitor.finalSchema();
9
10    return m.writerWithDefaultPrettyPrinter().
11        writeValueAsString(jsonSchema);
12 }
```

5.7 Auffälligkeiten beim Testen

Beim Serialisieren und Deserialisieren sollen Attribute in JSON-Dokumenten gespeichert werden. Hiefür benötigen alle Klassen einen Standard-Konstruktor damit der Serializer diesen 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 gewährleisten muss eventuell das OPM-Modell angepasst werden. Da alle Klassen sich an die OPM-Regeln halten, ist somit gewährleistet dass 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 vorgegeben.

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 ausführlich erläutert.

5.7.1 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. 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 Struktur 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 Rückgabotyp.

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. Bei den erstellten Klassen handelt es sich natürlich nur um einfache Gerüste, aber mehr wird für eine Serialisierung beziehungsweise Deserialisierung nicht benötigt. Die generierten Datenströme können 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 5.5 beschreiben. Auch das entsprechende Klassenschema soll vom GDS in einer Datenbank abgelegt werden.

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

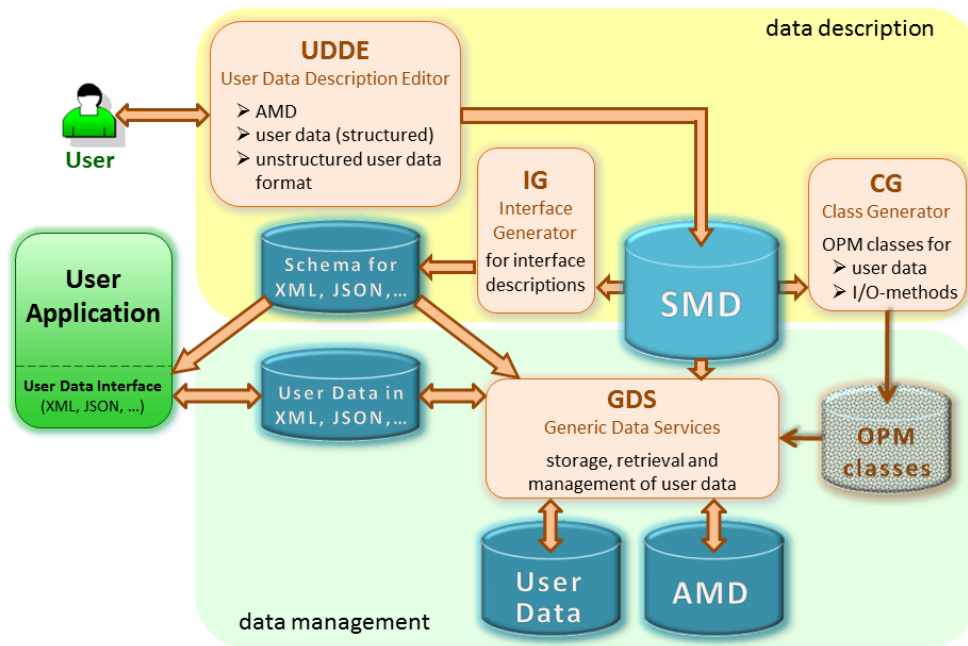


Abbildung 2: GDS Übersicht

6.1 Spezifikation des SMD-Assistent

Im Verlauf der Arbeit wurde zunehmend klar, dass 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 öfter 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` zusammengefasst 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.

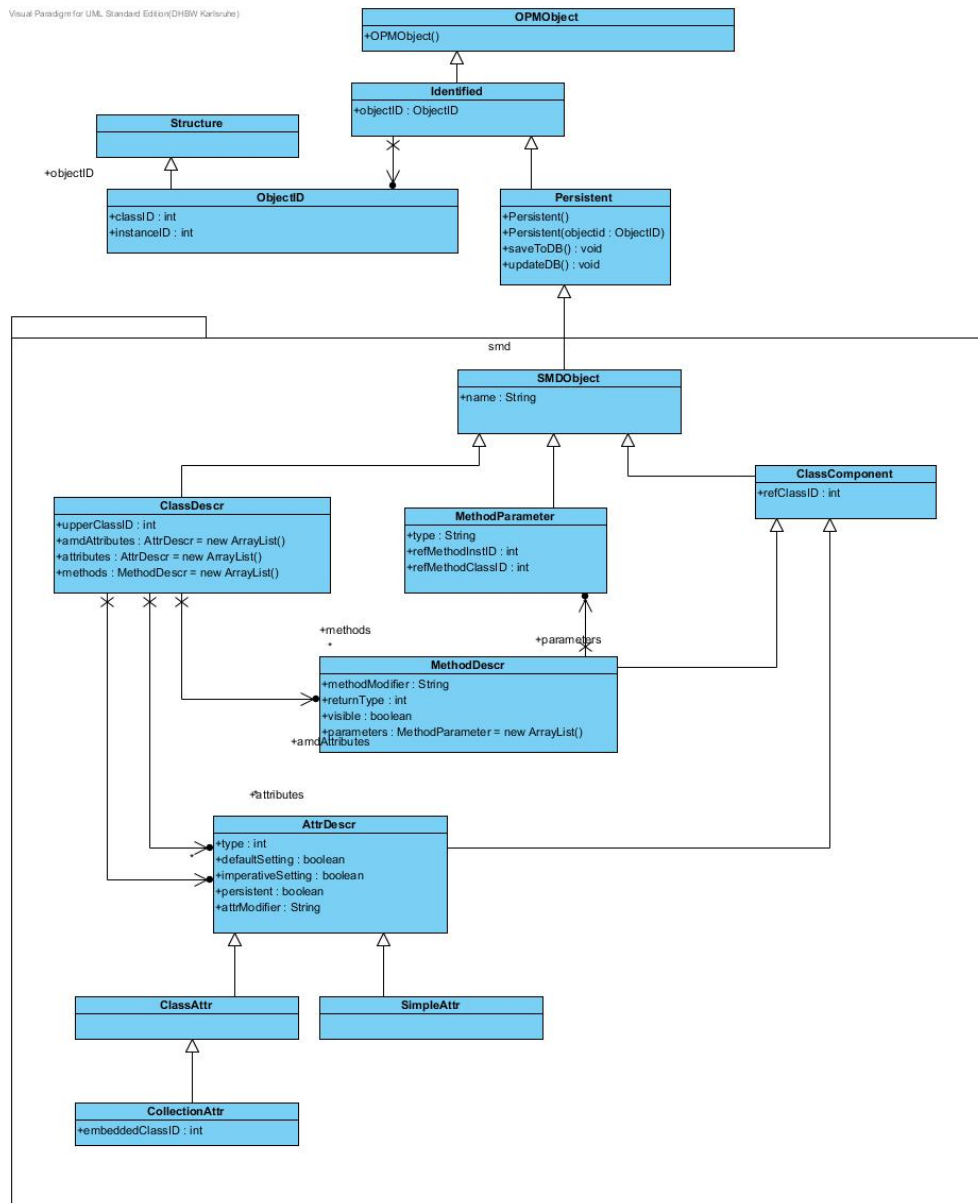


Abbildung 3: Klassendiagramm der SMD-Klassen

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.

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 Bibliotheken genauer zu beurteilen.

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 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 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 dritte Testklasse enthält 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 ein 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 durchzuführen und ihn jeweils drei mal zu wiederholen, wenn keine extremen Schwankungen auftreten. Sollte der Fall sein muss ein Test öfters wiederholt werden.

7.4.1 Auffälligkeiten bei Testbeginn

Die große Klasse mit einem Gibibyte Daten bereitete 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 basiert, 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 Millisekunden.

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 Testzusammenfassung und Auswertung

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 dass die Serialisierung eines einzelnen Elements quasi keine Zeit in Anspruch nimmt, den die Zeit für ein Element liegt bei $7,67111 \cdot 10^{-5}$ 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 darauffolgenden Parallelisierung hervorgerufen.

Welche Mechanismen wo genau ansetzten 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, dass mindestens vier mal so viel Ram benötigt wird, wie die Datei eigentlich groß ist.

8 Abkürzungsverzeichnis

KIT *Karlsruher Instituts für Technologie*

GDS *generisches Managementsystem für Energiedaten*

LSDF *Large Scale Data Facility*

OPM *objektorientierten Programmiermodell*

SMD *strukturelle Metadaten*

JSON *JavaScript Object Notation*

HALO *High Altitude and Long Range Research Aircraft*

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 / KIT. 2014. – Forschungsbericht
- [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 / KIT. 2014. – Forschungsbericht