Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Generierung von GeoServer-Styles für OHDM

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

1. Prüfer: Prof. Dr.-Ing. Thomas Schwotzer
2. Prüfer: Prof. Dr. Alexander Huhn

Eingereicht von Marcel Ebert

28. August 2019

Kurzfassung

In diesem Projekt soll eine Software zur Generierung von Dateien mit Styling-Informationen für einen *GeoServer* entwickelt werden. Ein *GeoServer* ist unter anderem dazu in der Lage Geodaten zu verarbeiten und daraus Kartenausschnitte zu erzeugen. Die erzeugten Dateien sollen von dem, im *OHDM*-Projekt eingesetzten, *GeoServer* benutzt werden, um die Darstellung der Geodaten zu konfigurieren, bevor diese in einer Karte gerendert werden. *OHDM* ist eine Abkürzung für „Open Historical Data Map“ und bezeichnet ein Projekt, das sich mit der Archivierung und Darstellung von historischen Kartendaten befasst.

In dieser Arbeit sollen Möglichkeiten erarbeitet werden, um die Generierung auf einfache Art und Weise konfigurieren zu können, vorzugsweise über eine einzige Konfigurationsdatei. Dort können die vorhandenen Typen von Geodaten gruppiert und den verschiedenen Gruppen einheitliche Styles zugewiesen werden. Weiterhin soll die Software eine automatisierte Konfiguration ermöglichen. Dabei sollen die generierten Dateien auf einem *GeoServer* an den richtigen Stellen platziert und der Server so konfiguriert werden, dass die neue Style-Konfiguration verwendet wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Akronyme

CRS Coordinate Reference System

CSS Cascading Style Sheets

JAR Java Archive

JSON JavaScript Object Notation

OGC Open Geospatial Consortium

OHDM OpenHistoricalDataMap

OSM OpenStreetMap

OWS OGC Web Services

PEG Parsing Expression Grammar

SLD Styled Layer Descriptor

TOML Tom's Obvious, Minimal Language

VPN Virtual Private Network

WCS Web Coverage Service

WFS Web Feature Service

WMS Web Map Service

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1 Einleitung

1.1 Hintergrund der Arbeit

Die Transformation von rohen Geodaten zu einer digitalen Karte ist ein wichtiger Bestandteil von Kartendiensten und Aufgabe von sogenannten Karten-Renderern. Um dies erreichen zu können benötigen die Renderer Informationen über die gewünschte Darstellung der Geodaten. Am weitesten verbreitet ist die Konfiguration der Visualisierung in verschiedenen Definitionsdateien, welche entsprechenden Geodatensätzen zugeordnet werden. Die große Menge an unterschiedlichen Klassen von Geodaten und deren individuelle Konfigurationsmöglichkeiten können zu unzähligen Konfigurationsdateien führen, bei welchen es schwierig werden kann, den Überblick zu behalten. Außerdem wird eine konsistente Pflege der Daten schwierig, wenn die Stile von ganzen Gruppen verändert werden sollen, da Änderungen an mehreren Dateien simultan stattfinden müssen.

1.2 Zielstellung

Möchte man das Aussehen bestimmter Gruppen von Geodaten in den gerenderten Karten des *OHDM*-Projektes verändern, bedarf es aktuell der manuellen Erzeugung bzw. Änderung von unzähligen Styling-Dateien. Ziel dieses Projektes ist es daher, dieses durch Automatisierung zu ersetzen und somit zu vereinfachen.

Wenn die Informationen über das gewünschte Aussehen der Geodaten an einem einzigen Ort verfügbar wären, müssten nur an dieser Stelle Änderungen gemacht werden, welche anschließend von der Software in die einzelnen Dateien überführt werden. Weiterhin können für nicht gesetzte Einstellungen günstig gewählte Standardwerte verwendet werden, um die Konfiguration für den Nutzer zu vereinfachen.

Das Ergebnis dieser Arbeit soll dazu beitragen, die Konfiguration von Karten-Renderern zu vereinfachen und eine konsistente Pflege der Daten zu erleichtern.

1.3 Abgrenzung

Obwohl die Generierung von Styles in dieser Arbeit eine große Rolle spielt, wird im Folgenden nur beispielhaft besprochen, wie deren Erzeugung mithilfe des entwickelten Systems funktioniert. Das bedeutet, dass die Erstellung von einsetzbaren bzw. sinnvollen Styles letztendlich Aufgabe des Anwenders dieser Software ist.

1.4 Aufbau der Arbeit

Zunächst werden die für das Verständnis der Inhalte dieser Arbeit erforderlichen Grundlagen erläutert. Darauf folgt die Erhebung und Analyse der Anforderungen an das zu entwickelnde System.

Im Kapitel über Konzeption und Entwurf wird eine Lösung skizziert, deren Umsetzung dann letztendlich im Kapitel „Implementierung“ beschrieben wird.

Der Prozess der Sicherstellung, dass das entwickelte System die erhobenen Anforderungen erfüllt, wird im Kapitel „Tests“ dargestellt.

Abschließend werden die Ergebnisse diskutiert und der Inhalt dieser Abschlussarbeit zusammengefasst.

2 Grundlagen

2.1 Open Historical Data Map

OpenHistoricalDataMap (OHDM) ist ein von Herrn Professor Dr. Schwotzer im Jahr 2015 an der HTW Berlin im Rahmen der Lehre gestartetes Projekt (vgl. [Sch17]). Die grundsätzliche Idee des Projektes ist die Sammlung bzw. das Speichern historischer Kartendaten, um auf Basis dieser Karten zu erzeugen, welche die zu einem bestimmten Zeitpunkt geltenden Geodaten darstellen.

Zur Erweiterung der Sammlung historischer Daten gibt es in *OHDM* zwei Szenarien: Die jährliche Archivierung der *OpenStreetMap (OSM)*-Daten und der Import weiterer Geometrien bzw. historischer Daten über andere Schnittstellen (vgl. [Sch17]). Die Karten sollen in Form von Bildern entstehen, welche auf Basis von, in einer *PostGIS*-Datenbank gespeicherten, Vektorgeometrien erzeugt werden (vgl. [Sch17]). Das Benutzungsszenario sieht dabei so aus, dass ein Nutzer einen Bereich der Weltkarte sowie ein beliebiges Datum angibt, woraufhin *OHDM*, basierend auf den gespeicherten historischen Daten, eine für diesen Zeitraum geltende Karte anzeigt (vgl. [Sch15]).

2.2 OpenStreetMap

OpenStreetMap ist ein Projekt mit dem Ziel eine freie, editierbare Weltkarte zu entwickeln. Der Zugriff auf die Karte bzw. die Geodaten ist dabei nahezu uneingeschränkt kostenlos möglich, um die Entwicklung interessanter neuer Möglichkeiten der Verarbeitung dieser Daten zu fördern (vgl. [Ope17]).

2.2.1 Map Features

Bei *OSM* gibt es sogenannte *Map Features*. Dieser Begriff bezeichnet Attribute, die Elementen einer Karte zugewiesen werden können, um deren physische Merkmale zu beschreiben (vgl. [Ope19j]). Obwohl es theoretisch möglich ist, diese Attribute beliebig zu wählen, werden Anstrengungen unternommen, eine ausgewählte Liste von den am häufigsten benutzten Tags in Form eines informellen Standards zu pflegen (vgl. [Ope19j]). Diese Liste kann im *OSM*-Wiki[1] eingesehen werden.

2.2.2 Zoomstufen

Weiterhin definiert das *OSM*-Wiki (in [Ope19k]) einen Vorschlag für eine Einteilung der Karte in 20 verschiedene Zoomstufen. Diese Einteilung dient als Orientierung und bietet Beispiele dafür, welche Inhalte auf einer bestimmten Zoomstufe gerendert werden sollten.

2.3 Styled Layer Descriptor

Styled Layer Descriptor (SLD) ist die Bezeichnung eines von der *Open Geospatial Consortium (OGC)* spezifizierten Kodierungs-Standards, welcher definiert, wie ein *Web Map Service (WMS)* erweitert werden kann, um benutzerdefiniertes Styling zu ermöglichen (vgl. [Ope19a]). Dieser Standard soll sowohl für Nutzer als auch Software eine Möglichkeit schaffen, durch Verwendung einer für Menschen und Maschinen lesbaren Sprache die visuelle Repräsentation von Geodaten zu kontrollieren (vgl. [Ope19a]).

Das Format dieses Standards wird in der „Symbology Encoding Implementation Specification“[2] der *OGC* beschrieben und soll hier nicht im Detail besprochen werden.

2.4 GeoServer

GeoServer bezeichnet eine in Java geschriebene quelloffene Serveranwendung, welche Benutzern das Teilen und Bearbeiten von Geodaten ermöglicht (vgl. [Ope19f]).

Das Projekt stellt eine Referenzimplementierung der, vom *OGC* spezifizierten, *Web Feature Service (WFS)*[3]-, *Web Coverage Service (WCS)*[4] und *WMS*[5]-Standards dar (vgl. [Ope19f]).

[1] Liste der Map Features: https://wiki.openstreetmap.org/wiki/Map_Features
[2] Definition der Spezifikation: http://www.opengeospatial.org/standards/se
[3] Definition des WFS-Standards siehe: http://www.opengeospatial.org/standards/wfs
[4] Definition des WCS-Standards siehe: http://www.opengeospatial.org/standards/wcs
[5] Definition des WMS-Standards siehe: http://www.opengeospatial.org/standards/wms

2.4.1 Terminologie

GeoServer verwendet verschiedene Objekttypen, um die interne Struktur abzubilden. Eine Auflistung und Beschreibung der wichtigsten Objekte ist in Tabelle 2.1 abgebildet.

***GeoServer* Objekte**	**Beschreibung**
Workspace	Ein Container, der mehrere (meist ähnliche) Layer gruppiert (vgl. [Ope19g]).
Namespace	Eng gekoppelt mit einem Workspace. Enthält ein Prefix, das dem Namen des Workspaces entspricht und eine URI, die eine passende URL sein kann, aber nicht zwingenderweise zu einer existierenden Web-Adresse auflösen muss (vgl. [Ope19g]).
Store	Enthält Informationen über eine Datenquelle, die Zugriff auf Geodaten gewährt. Dabei kann die Datenquelle eine (Gruppe von) Datei(en), eine Datenbanktabelle, eine Rasterdatei oder ein Verzeichnis sein (vgl. [Ope19h]).
Layer	Ein Raster- oder Vektordatensatz, der eine Sammlung von geografischen Daten repräsentiert (vgl. [Ope19e]).
FeatureType	Eine vektorbasierte Ressource oder ein Datensatz, welcher Eckdaten wie das Koordinatenreferenzsystem, Bounding-Boxen und Projektion spezifiziert (vgl. [Ope19d]).
StyleInfo	Enthält Metadaten für einen Style. Dazu gehören Name, Dateiname und Format.

Tabelle 2.1: Übersicht der wichtigsten *GeoServer*-Objekte

2.4.2 Extensions

Die Installation von Extensions, also Modulen, welche die Anwendung um zusätzliche Funktionalitäten erweitern, wird von *GeoServer* unterstützt.

Eine dieser Erweiterungen ist die *CSS-Extension*. Diese ermöglicht die Definition von *Styled Layer Descriptors* in einer von *Cascading Style Sheets (CSS)* abgeleiteten Sprache (vgl. [Ope19c]). Die auf diese Art definierten Styles werden von der Extension in das herkömmliche *SLD*-Format umgewandelt und können danach vom *GeoServer* verwendet werden (vgl. [Ope19c]).

2.5 Parsing Expression Grammars

Parsing Expression Grammars (PEGs) wurden im Jahr 2004 von Bryan Ford (in [For04]) vorgestellt und stellen eine Alternative zu kontextfreien Grammatiken dar.

2.5.1 Ziel

Das Problem kontextfreier Grammatiken ist, dass sie Mehrdeutigkeiten zulassen. Dies ist für die Modellierung natürlicher Sprachen geeignet, kann bei maschinenorientierten Sprachen allerdings zu Problemen führen (vgl. [For04, S. 1]). Um dieses Problem zu umgehen sind Mehrdeutigkeiten in *Parsing Expression Grammars* nicht definierbar. Das wird unter anderem dadurch erreicht, dass in *Parsing Expression Grammars* anstelle von „nicht-deterministischer Wahl“ „priorisierte Wahl“ eingesetzt wird. (vgl. [For04, S. 1]).

2.5.2 Definition von Parsing Expression Grammars

Eine *PEG* wird durch ein 4-Tupel $G = (V_N, V_T, R, e_S)$ definiert, wobei V_N eine endliche Menge nicht-terminaler Symbole, V_T eine endliche Menge terminaler Symbole, R eine endliche Menge von Regeln, e_S der „start expression“ Parse-Ausdruck und $V_N \cap V_T = \emptyset$ ist (vgl. [For04, S. 4]).

Für die Definition der Regeln existiert eine bestimmte Menge an Operatoren. Diese sind in Tabelle 2.2 abgebildet.

Operator	Type	Precedence	Description
‘ ’	primary	5	Literal string
“ ”	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(e)	unary suffix	4	Grouping
e?	unary suffix	4	Optional
e*	unary suffix	4	Zero-or-more
e+	unary suffix	4	One-or-more
&e	unary suffix	3	And-predicate
!e	unary suffix	3	Not-predicate
$e_1 e_2$	binary	2	Sequence
e_1 / e_2	binary	1	Prioritized Choice

Tabelle 2.2: Operatoren zur Konstruktion von Parse-Ausdrücken (Tabelle aus [For04, S. 3])

Weitere Informationen, wie z. B. die Definition der Regeln und Ausdrücke, können Bryan Ford’s Publikation[6] entnommen werden.

[6]Link zur Publikation: http://bford.info/pub/lang/peg.pdf

3 Anforderungserhebung und -analyse

In den folgenden Abschnitten werden die erhobenen Benutzeranforderungen definiert. Jede Anforderung ist, sofern nicht anders angegeben, als obligatorisch zu behandeln.

3.1 Funktionale Anforderungen

Die Gruppierung der funktionalen Anforderungen basiert auf der Zugehörigkeit zu einem Themenbereich.

3.1.1 Eingabeparameter

Ein Benutzer soll dem Programm Parameter übergeben können, welche Einfluss auf den Ablauf des Programms nehmen.

Ein Benutzer soll den Pfad zu einer *OHDM*-Konfigurationsdatei als Eingabeparameter übergeben können. Diese soll die Anweisungen zur Generierung der Styles enthalten. Die Angabe dieser Datei soll obligatorisch sein.

Ein Benutzer soll den Pfad zu einer *GeoServer*-Konfigurationsdatei als Eingabeparameter übergeben können. Diese Datei soll Angaben über den zu konfigurierenden *GeoServer* beinhalten.

Ein Benutzer soll den Pfad zu einer Datenbank-Konfigurationsdatei als Eingabeparameter übergeben können. Diese Datei soll Informationen zu Verbindungsparametern einer Datenbank beinhalten.

Ein Benutzer soll den Pfad zu einem Ausgabeverzeichnis als Eingabeparameter übergeben können. Das System soll die generierten *SLD*-Dateien in diesem Verzeichnis platzieren. Falls keine Angabe gemacht wird, soll das aktuelle Verzeichnis als Ausgabeverzeichnis gewählt werden.

Fakultativ Ein Benutzer soll dem Programm einen Parameter übergeben können, welcher entscheidet, ob bei der *OHDM*-Konfigurationsdatei nicht berücksichtigte Klassen mit Standardwerten aufgefüllt werden sollen.

3.1.2 OHDM-Konfigurationsdatei

Die Konfigurationsdatei soll die Gruppierung von Klassen und deren Subklassen unterstützen. Diese Gruppierung wird fortan Klassen-Gruppierung bzw. *Map Feature* (in Anlehnung an den in Abschnitt 2.2.1 beschriebenen *OSM*-Begriff) genannt. Den Subklassen sollen verschiedene Styles zugewiesen werden können. Dabei soll es möglich sein, verschiedene Zoombereiche festzulegen, in welchen bestimmte Styles angewendet werden.

Für die Definition der Zoombereiche soll eine sinnvolle Einteilung in Zoomstufen festgelegt werden.

Die Konfigurationsdatei soll die Verlinkung von Klassen-Gruppierungen unterstützen. Das bedeutet, dass eine Klassen-Gruppierung auf eine andere verweisen kann und dadurch, sofern möglich, dieselben Styles erhält.

Fakultativ Die Definition von Styles soll auch ausserhalb der Klassen-Gruppierungen möglich sein. Darüber hinaus soll es die Möglichkeit geben, innerhalb der Klassen-Gruppierungen auf diese Styles zu referenzieren. Dadurch sollen Styles an mehreren Stellen wiederverwendet werden, ohne eine Duplizierung zu erfordern.

3.1.3 Automatisierte Konfiguration

Das System soll bei Existenz einer *GeoServer*- und Datenbank-Konfiguration alle Konfigurationsdateien generieren, die für die automatisierte Konfiguration des *GeoServers* nötig sind.

Das System soll die generierten Konfigurationsdateien nicht in dem vom Benutzer bestimmten Ausgabeverzeichnis platzieren (weil dort nur die *SLD-Dateien* platziert werden sollen).

Das System soll bei Angabe einer *GeoServer*-Konfigurationsdatei versuchen, die generierten Konfigurationsdateien gemäß den Angaben in der Konfiguration, auf dem angegebenen Server zu platzieren.

3.2 Nichtfunktionale Anforderungen

Die folgenden nichtfunktionalen Anforderungen werden entsprechend ihrer Eigenschaften den „Arten nichtfunktionaler Anforderungen“ (nach Ian Sommerville [Som12, S. 120]) zugeordnet.

3.2.1 Zuverlässigkeitsanforderungen

Ein Benutzer soll im Fehlerfall detaillierte Informationen über das Problem erhalten.

Die Wahrscheinlichkeit einer Datenzerstörung bei Eintritt einer Fehlfunktion soll minimiert werden.

Fakultativ Einem Benutzer sollen im Fehlerfall Hilfeanweisungen zur Problemlösung angezeigt werden.

3.2.2 Umgebungsanforderungen

Die Software soll plattformunabhängig funktionieren. Das bedeutet, dass das Programm auf den gängigen Plattformen ausführbar sein muss und es irrelevant ist, auf welchem Betriebssystem der *GeoServer* betrieben wird.

Die Software soll unabhängig von der eingesetzten *GeoServer* Version funktionieren.

3.2.3 Benutzbarkeitsanforderungen

Ein Benutzer soll die Möglichkeit haben, die vorhandenen Optionen zum Programmaufruf von der Software zu erfragen.

Das für die Konfiguration der Styles eingesetzte Dateiformat soll für Menschen lesbar sein.

Die Syntax des für die Konfiguration der Styles eingesetzten Dateiformates ist intuitiv und der Umgang leicht zu erlernen.

Das System soll Möglichkeiten bieten, eine automatisierte Konfiguration eines *GeoServers* mit den generierten *SLD*-Dateien durchzuführen. Das bedeutet, dass der *GeoServer* nach Durchlauf des Programmes die neuen Styles für das Styling der Geodaten benutzen soll.

Ein Benutzer soll textuelle Informationen über den Ablauf bzw. aktuellen Fortschritt des Programms in Form von Ausgaben auf der Konsole erhalten.

4 Konzeption & Entwurf

4.1 Wahl der Programmiersprache

In diesem Projekt soll zur Entwicklung der Anwendung die Programmiersprache *Java* verwendet werden. Im Folgenden werden die Gründe für diese Entscheidung dargestellt.

Die Umgebungsanforderungen in Abschnitt 3.2.2 schreiben vor, dass die Software plattformunabhängig funktionieren soll. *Java* ist aufgrund seiner Architektur auf jeder Plattform einsetzbar, die eine *Java Runtime Environment* zur Verfügung stellt. Das bedeutet, dass die Endanwendung ohne Veränderungen am Quelltext auf vielen verschiedenen Systemen funktioniert.

Weiterhin wird der *GeoServer* größtenteils in *Java* entwickelt. Da die Benutzung der *GeoServer*-Module für die Konfiguration von größerer Relevanz ist, ist die Verwendung der gleichen Programmiersprache vorteilhaft und konsequent.

4.2 Die OHDM-Konfigurationsdatei

4.2.1 Wahl des Dateiformates

Um die an die *OHDM*-Konfigurationsdatei gestellten Anforderungen zu erfüllen, muss ein geeignetes Dateiformat gewählt werden. Hierbei stehen diverse Möglichkeiten zur Verfügung. Für eine bessere Übersicht wurde eine Auswahl unterschiedlicher Dateiformate in Tabelle 4.1 gegenübergestellt. Es werden die Vor- und Nachteile von *Java-Properties*[1], *JavaScript Object Notation (JSON)*[2], *Extensible Markup Language (XML)*[3], *YAML Ain't Markup Language (YAML)*[4], *Tom's Obvious, Minimal Language (TOML)*[5] und einem eigenen, projektspezifischen Dateiformat betrachtet.

[1] Link zur Properties-Spezifikation: https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html
[2] Link zur JSON-Spezifikation: https://tools.ietf.org/html/rfc8259
[3] Link zur XML-Spezifikation: https://www.w3.org/TR/xml/
[4] Link zur YAML-Spezifikation: https://yaml.org/spec/1.2/spec.html
[5] Link zum TOML-Github-Projekt: https://github.com/toml-lang/toml

Dateiformat	Vorteile	Nachteile
Java-Properties	Native Java-Unterstützung	Veraltet und nicht so weit verbreitet Unterstützt nur Key-Value Paare
JSON	Weit verbreitet Leicht verständlich	Keine Kommentare möglich Unübersichtlich bei komplexen Strukturen
XML	Weit verbreitet Leicht verständlich	Datendefinition sehr ausschweifend Unübersichtlich bei komplexen Strukturen
YAML	Leicht lesbar Relativ leicht zu editieren	Sicherheitslücken in Parsern bekannt Unübersichtlich bei komplexen Strukturen
TOML	Einfache Syntax Leicht zu lesen/schreiben Kommentare möglich	Relativ jung Hohe Redundanz bei tiefer Schachtelung der Daten Noch in Pre-Release Phase
Eigenes	Anpassung auf Projektanforderungen Gestaltungsfreiheiten	Höherer Initialaufwand Kein etablierter Standard

Tabelle 4.1: Vor- und Nachteile ausgewählter Dateiformate

Die Entscheidung basiert auf der Beurteilung der Anforderungen, die in Abschnitt 3.1.2 und Abschnitt 3.2.3 definiert wurden. Die hierfür relevanten Anforderungen sind im Wesentlichen, dass das Dateiformat für Menschen lesbar, die Syntax intuitiv und der Umgang leicht zu erlernen, die Verlinkung von Gruppierungen unterstützt, und eine, von den Klassen-Gruppierungen unabhängige, Definition der Styles möglich sein sollen.

Generell sind alle Dateiformate für Menschen lesbar. Der Umgang ist dabei unterschiedlich intuitiv, vor allem bei *YAML* und *TOML* ist eine längere Eingewöhnungszeit zu erwarten.

Aufgrund der sehr einfach zu verstehenden Syntax und der Möglichkeit für Kommentare, sowie einer Unterstützung komplexerer Datenstrukturen fiel die Wahl zuerst auf *TOML*. Eine weitere Evaluierung ergab allerdings, dass die Daten durch Angabe von Klasse, Subklasse und Zoomstufe relativ tief verschachtelt werden und die Definition bei Verschachtelung in *TOML* sehr redundant wird. Deshalb wurde sich für den Entwurf eines eigenen Parsings entschieden.

Die beste Balance zwischen Aufwand und Erfüllung der Anforderungen lässt sich durch den Entwurf eines eigenen Dateiformates erreichen. Dadurch entstehen Möglichkeiten, die Verlinkungen zwischen den Gruppierungen intuitiv zu gestalten während Redundanz weitgehend vermieden werden kann. Zum Beispiel kann durch Verwendung besonderer Zeichen zwischen der Definition einer Klassen-Gruppierung und einem separaten Style unterschieden werden. Weiterhin wird dadurch die Integration der in Abschnitt 2.4.2 beschriebenen *CSS*-Extension ermöglicht.

4.2.2 Entwurf des Formats

Beim Entwurf des Formats wurden die Anforderungen aus Abschnitt 3.1.2 berücksichtigt. Die Syntax wurde an *JSON* angelehnt, wobei diverse Vereinfachungen implementiert wurden. Als Dateiendung wird, um den Kontext der Datei zu signalisieren, die Benutzung von „.ohdmconfig“ empfohlen, diese ist allerdings nicht zwingend erforderlich. Im Folgenden wird der Aufbau schrittweise anhand des Beispiels in Listing 4.1 erläutert.

Über die Zeilen 1-18 erstreckt sich die Definition eines *Map Features*. In Zeile 1 wird der Name des *Map Features* angegeben und der Körper für die Definition der Eigenschaften geöffnet. Das *Map Feature* mit dem Name „aeroway“ bekommt in diesem Beispiel Subklassen mit den Namen „undefined“ (definiert in den Zeilen 2-6) und „aerodrome“ (definiert in den Zeilen 7-17) zugeordnet. Die Anweisungen für das Styling müssen von einem Zoombereich umschlossen sein.

Es gibt folgende Möglichkeiten einen Zoombereich zu definieren:

- Durch Angabe eines Bereiches, wie z. B. „5-10“,
- durch Angabe einer oberen oder unteren Grenze, also „<10“ oder „>10“,
- oder durch Angabe des „default“-Schlüsselwortes.

Bei Angabe von „<10“ wird der Wert der Zoomstufe 10 als maximale Map-Skalierung festgelegt, bei welcher der Style gerendert werden soll. Das bedeutet, er wird dann bei Skalierungen gerendert, welche kleiner als der Wert von Zoomstufe 10 sind. Gleichermaßen wird bei Festlegung des Zoombereiches auf „>10“ der Wert der Zoomstufe 10 als minimale Map-Skalierung festgelegt, sodass der Style bei Skalierungen, welche größer als der Wert von Zoomstufe 10 sind, angewendet wird.

Bei Benutzung von „default“ wird für die Regel kein spezifischer Zoombereich festgelegt.

Innerhalb des Körpers der Zoombereich-Definition werden die Anweisungen für das Styling hinterlegt. Dabei kann entweder auf separate Styles (`StyleGroups`) referenziert oder der Style durch Benutzung der Syntax der, in Abschnitt 2.4.2 vorgestellten, *CSS*-Extension definiert werden. Die Referenzen werden (wie in Zeile 4 zu sehen) mithilfe des Schlüsselwortes „useStyle“ zugewiesen. Anweisungen für die *CSS*-Extension werden ohne Schlüsselwort platziert.

In den Zeilen 20-22 wird ein weiteres *Map Feature* definiert. Dieses referenziert das vorherige *Map Feature*, um die selben Regeln zugewiesen zu bekommen, ohne die Anweisungen zu duplizieren. Der Verweis wird mit „sameAs“ eingeleitet und ist in Zeile 21 zu sehen.

`StyleGroups` werden im Gegensatz zu *Map Features* mit spitzen Klammern („<...>“) eingeleitet. Bei diesen ist es nur möglich die Style Anweisungen mithilfe der *CSS*-Extension zu definieren. Dafür können sie von den *Map Features* referenziert werden.

```
[aeroway] {
    [undefined] {
        [default] {
            useStyle = SimplePolygon, SimplePoint, SimpleLine
        }
    }
    [aerodrome] {
        [0-10] {
            * {
                mark: symbol(circle);
                mark-size: 6px;
            }
            :mark {
                fill: red;
            }
        }
    }
}

[waterway] {
    sameAs = aeroway
}

<SimplePolygon> {
    * {
        fill: #000080;
        fill-opacity: 0.5;
        stroke: #FFFFFF;
        stroke-width: 2;
    }
}

<SimplePoint> {
    * {
       mark: symbol(triangle);
       mark-size: 12;
       :mark {
         fill: #009900;
         fill-opacity: 0.2;
         stroke: black;
         stroke-width : 2px;
       }
    }
}

<SimpleLine> {
    * {
        stroke: blue;
        stroke-width: 3px;
        stroke-dasharray: 5 2;
    }
}
```

Listing 4.1: Simple OHDM-Konfigurationsdatei

4.2.3 Definition der Grammatik für die OHDM-Konfigurationsdatei

Die Grammatik für die *OHDM*-Konfigurationsdatei wurde auf Basis des Entwurfs aus Abschnitt 4.2.2 entwickelt. Die Definition richtet sich nach den in Abschnitt 2.5.2 vorgestellten Regeln.

Die Grammatik ist definiert als $G = (V_N, V_T, R, e_S)$, wobei

V_N = { OHDMConfig, MapFeatureDeclaration, MapFeatureSubclassDeclaration, MapFeatureDeclaration, MapFeatureReferenceDeclaration, MapFeatureReferenceName, SubclassRuleDeclaration, ZoomRegionDeclaration, StyleGroupDeclaration, StyleDeclaration, StylePlaceholderDeclaration, DetailedStyleDeclaration, ***StyleSheet***, MapFeatureName, StyleGroupName, SubclassName, StyleNames, ZoomNumber, String, NonReservedCharacter, EqualsSymbol, BeginVariableSymbol, EndVariableSymbol, BeginBlockSymbol, EndBlockSymbol, AlphabeticCharacter, AlphanumericCharacter, Digit, OptionalWhiteSpace, EOI } ,

V_T = { [,], <, >, {, }, -, _, =, [A..Z], [a..z], [0..9], default, sameAs, useStyle } ,

R = {

OHDMConfig	←	MapFeatureDeclaration* StyleGroupDeclaration* EOI
MapFeatureDeclaration	←	MapFeatureName BeginBlockSymbol MapFeatureReferenceDeclaration / MapFeatureSubclassDeclaration+ EndBlockSymbol
MapFeatureName	←	BeginVariableSymbol String EndVariableSymbol
MapFeatureReferenceDeclaration	←	'sameAs' OptionalWhiteSpace EqualsSymbol OptionalWhiteSpace MapFeatureReferenceName OptionalWhiteSpace
MapFeatureReferenceName	←	String
MapFeatureSubclassDeclaration	←	SubclassName BeginBlockSymbol SubclassRuleDeclaration+ EndBlockSymbol
SubclassName	←	BeginVariableSymbol String EndVariableSymbol
SubclassRuleDeclaration	←	ZoomRegionDeclaration BeginBlockSymbol StyleDeclaration EndBlockSymbol
ZoomRegionDeclaration	←	OptionalWhiteSpace BeginVariableSymbol (ZoomNumber '-' ZoomNumber) / ('>' ZoomNumber) / ('<' ZoomNumber) EndVariableSymbol OptionalWhiteSpace
ZoomNumber	←	Digit+
StyleDeclaration	←	OptionalWhiteSpace StylePlaceholderDeclaration / DetailedStyleDeclaration OptionalWhiteSpace
StylePlaceholderDeclaration	←	'useStyle' OptionalWhiteSpace EqualsSymbol OptionalWhiteSpace StyleName
StyleNames	←	((String OptionalWhiteSpace ',' OptionalWhiteSpace)+ String) / String
DetailedStyleDeclaration	←	StyleSheet
StyleSheet	←	***(siehe CSS-Extension)***
String	←	AlphabeticCharacter NonReservedCharacter*
AlphabeticCharacter	←	[a..z] / [A..Z]
NonReservedCharacter	←	AlphanumericCharacter / ('-' / '_')
AlphanumericCharacter	←	AlphabeticCharacter / Digit
Digit	←	[0..9]
EqualsSymbol	←	'='
BeginBlockSymbol	←	OptionalWhiteSpace '{' OptionalWhiteSpace
EndBlockSymbol	←	OptionalWhiteSpace '}' OptionalWhiteSpace
BeginVariableSymbol	←	'['
EndVariableSymbol	←	']'
OptionalWhiteSpace	←	(' ' / '\r' / '\t' / '\f' / '\n')*
EOI	←	!.

}

und e_S = OHDMConfig.

Die Aufschlüsselung der terminalen und nichtterminalen Symbole sowie Regeln, die von der *CSS-Extension* benutzt werden, wurde aus mehreren Gründen weggelassen. Einerseits handelt es sich nicht um eigenes Gedankengut, andererseits ist die Anzahl der Regeln beträchtlich und für das Verständnis der Grammatik dieses Dateiformates weitgehend irrelevant. Man kann sich vereinfacht vorstellen, dass an die Stelle eines *StyleSheets* sämtliche Style-Definitionen treten können, welche die syntaktischen Anforderungen der *CSS*-Extension erfüllen[6].

4.3 Konfiguration des *GeoServers*

Die automatisierte Konfiguration des *GeoServers* mit den generierten *SLD*-Dateien stellt einen wichtigen Teil dieser Arbeit dar. Es ist zwar prinzipiell möglich die Software nur für die Generierung der *Styled Layer Descriptors* zu verwenden und diese anschließend manuell (über das Web-Interface) auf den *GeoServer* hochzuladen. Dieser Ansatz würde allerdings nur dann Sinn ergeben, wenn nur wenige ausgewählte Styles verwendet werden sollen. Um den repetitiven, manuellen Aufwand zu vermeiden stellt diese Software automatisierte Konfigurationsmöglichkeiten zur Verfügung, die im Folgenden genauer erläutert werden.

4.3.1 Architektur des *GeoServers*

Im Installationsverzeichnis des *GeoServers* existieren verschiedenste Unterverzeichnisse und Konfigurationsdateien im *XML*-Dateiformat, welche bei Ausführung des „startup.sh“-Skripts vom *GeoServer* geladen werden. Informationen zu den für dieses Projekt relevanten Einstellungen werden kurz zusammengefasst.

Das Hauptverzeichnis der Daten

Alle konfigurierbaren Informationen befinden sich in Unterverzeichnissen des „data_dir“-Verzeichnisses im Ursprung des *GeoServer*-Installationsverzeichnisses. Eine Auflistung der wichtigsten Unterverzeichnisse ist in Tabelle 4.2 dargestellt.

[6] Beispiele für Styles beschrieben in: https://docs.geoserver.org/master/en/user/styling/css/examples/index.html

Verzeichnisname	Funktion
data	Enthält Datenquellen für Geodaten, die nicht in Datenbanken abgebildet werden. Dazu gehören unter anderem „Shapefiles“, „Properties“- oder „WorldImage“-Dateien.
styles	Enthält die *SLD*-Dateien sowie deren zugehörige Konfigurationsdateien.
workspaces	Enthält sämtliche Informationen zu den eingerichteten *Workspaces*.

Tabelle 4.2: Unterverzeichnisse von „data_dir“ und deren Funktion

Das „styles“-Verzeichnis

Im „styles“-Verzeichnis werden die Dateien abgelegt, welche Styling-Anweisungen enthalten. Für jede dieser Dateien existiert eine zugehörige *XML*-Datei, welche Meta-Informationen beinhaltet. Zu diesen Informationen gehören unter anderem eine ID, der Name des Styles, das Format (z. B. *SLD* oder *CSS*) und der Name der Style-Datei für die diese Informationen gelten. Über die ID kann an anderen Stellen (wie z. B. bei Konfiguration eines Layers) auf einen Style verwiesen werden.

Das „workspaces“-Verzeichnis

Dieses Verzeichnis enthält für jeden angelegten *Workspace* ein Unterverzeichnis sowie eine Konfigurationsdatei, welche den Standardworkspace festlegt.

Ein Verzeichnis eines konkreten *Workspaces* beinhaltet Meta-Informationen über sich selbst, sowie Informationen zu dem dort geltenden *Namespace*. Weiterhin werden dort die Daten der eingerichteten *Stores* abgelegt.

Die *Store*-Verzeichnisse

Die in einem Workspace abgelegten *Store*-Verzeichnisse enthalten eine Datei mit Informationen über die Datenquelle, wie z. B. ID, Name, Typ und Verbindungsparametern sowie Unterverzeichnisse für jede eingerichtete *Layer*. In diesen Unterverzeichnissen befinden sich schliesslich Informationen über den *FeatureType*, welcher von der *Layer* dargestellt wird, wie auch eine Zuordnung des Styles zu selbiger.

4.3.2 Erforderliche Konfigurationsdateien

Aus den Informationen des vorherigen Abschnittes kann man ableiten, welche Konfigurationsdateien automatisiert erzeugt werden müssen. Konkret handelt es sich dabei also um Dateien für: *Styles*, *Workspaces*, *Namespaces*, *Stores*, *Layers* und *FeatureTypes*. Diese müssen so aufeinander abgestimmt werden, dass sie die korrekten Zuweisungen beschreiben.

5 Implementierung

5.1 Build-Management

5.1.1 Verwendetes Tool

Dieses Projekt verwendet „Gradle“ als Build-Management Werkzeug, um das Verwalten von Abhängigkeiten und die Erzeugung von Artefakten der Software zu vereinfachen.

Es wurde sich für den Einsatz von *Gradle* entschieden, da die Konfiguration durch den Einsatz von *Groovy* anstelle von *XML* prägnanter und weniger ausschweifend ist. Ausserdem ist *Gradle* performanter[1] und bietet mehr Möglichkeiten für Dependency-Management.

5.1.2 Besonderheiten

Bis auf eine Java Archive (JAR)-Datei für das *Main*-Modul des *GeoServers* werden alle Abhängigkeiten aus Maven-Repositories bezogen. Das *Main*-Modul des *GeoServers* muss separat eingebunden werden, da die Maven-Dependency Konflikte mit anderen Abhängigkeiten verursacht. Die *JAR*-Datei enthält eine abgeänderte Version einer Dependency, welche in dieser Form nicht auf Maven gehostet wird.

Weiterhin müssen, obwohl sie in diesem Projekt nicht direkt zur Anwendung kommen, Abhängigkeiten des „Spring“-Frameworks[2] geladen werden. Diese werden intern von der Implementierung des *GeoServers* verwendet; bei Abwesenheit der Dependencies treten Fehler bei der Erzeugung von Klassen des *GeoServers* auf.

[1] Siehe Darstellung unter „Performance“ bei https://gradle.org/maven-vs-gradle/
[2] Ein Framework für Java-Webanwendungen

5.1.3 Erzeugung eines ausführbaren Artefakts

Das ausführbare Artefakt soll in Form einer *JAR*-Datei erzeugt werden. Dazu wird auf das „shadow"-*Gradle*-Plugin[3] von John Rengelman zurückgegriffen. Dieses Plugin vereinfacht die Erzeugung von sogenannten „fat-JARs", also *JAR*-Dateien, welche alle von der Software benötigten Abhängigkeiten beinhalten. Durch das Hinzufügen des Plugins zur *Gradle*-Build-Konfiguration wird automatisch ein entsprechender *Gradle*-Task angelegt.

Um eine ausführbare JAR-Datei zu erzeugen muss `./gradlew shadowJar` im Root-Verzeichnis des Projektes ausgeführt werden. Die Datei wird im `build/libs`-Verzeichnis abgelegt.

5.1.4 Aufteilung in verschiedene Module

Obwohl es auf den ersten Blick sinnvoll erscheinen mag, die Teilaufgaben in unterschiedliche Module auszulagern und schlussendlich zwei verschiedene Artefakte zu erzeugen, wurde sich bewusst gegen diese Herangehensweise entschieden. Die Module würden bei diesem Szenario so aufgeteilt, dass das eine die Generierung der *SLD*-Dateien übernimmt und das andere die Konfiguration und den Upload der Dateien. Die Entscheidung gegen die Aufteilung ist primär damit begründet, dass es notwendig ist, schon bei der Generierung der *Styled Layer Descriptors* eine *Classification* zur Verfügung zu haben, welche bei Angabe einer Datenbank-Konfiguration aus dieser abgeleitet werden kann. Das Verzeichnis, das die *SLD*-Dateien beinhaltet, sowie die Datenbank-Konfiguration müssten auch dem anderen Module mitgeteilt werden. Beides sind Informationen, welche dem anderen Modul in den meisten Fällen bereits zur Verfügung stehen, sodass eine Aufteilung dieser Teilbereiche zusätzlichen Konfigurationsaufwand für den Nutzer bedeutet. Ausserdem ist davon auszugehen, dass der Nutzer im Regelfall eine sofortige und automatisierte Übertragung und Konfiguration der generierten Styles beabsichtigt.

5.2 Classification

Eine wichtige Rolle für das korrekte Setup spielt die *Classification*. Dabei handelt es sich um ein Objekt, welches Informationen zu den verwendbaren Klassen und deren Subklassen enthält.

Falls Angaben zu einer zu verwendenden Datenquelle vorhanden sind, wird versucht die *Classification* anhand dieser zu erzeugen. Dazu wird in dem angegebenen Schema nach einer Relation mit dem Namen `classification` gesucht, welche die Attribute `class` und `subclassname` besitzt. Auf Basis der Datensätze in dieser Tabelle wird ein neues `Classification`-Objekt erzeugt.

[3] GitHub-Seite des Projekts: https://github.com/johnrengelman/shadow

Falls keine Angaben zu einer Datenquelle gemacht wurden oder diese fehlerhaft sind, bzw. die `classification`-Relation nicht existiert, wird auf eine Standardimplementierung zurückgegriffen. Diese basiert auf der `OSMClassification`-Klasse des „OSMImportUpdate“-Projektes[4], wo zur Erzeugung einer `OSMClassification` die einzelnen von *OSM* definierten *Map Features* mit ihren möglichen Tags kombiniert werden.

Die *Classification* wird beim Parsen der übergebenen *OHDM*-Konfigurationsdatei verwendet, um den Nutzer über nicht bzw. falsch verwendete Klassen und deren Subklassen zu informieren und um Standarddateien für die nicht gesetzten Klassen zu generieren. Weiterhin wird sie verwendet, um die notwendigen Tabellen für die Geodaten zu erzeugen, falls diese noch nicht existieren (wird in Abschnitt 5.6.2 beschrieben).

5.3 Commandline-Interface

5.3.1 Verwendetes Framework

Um die Arbeit mit der Kommandozeile zu vereinfachen wird in diesem Projekt ein Framework eingesetzt. Für *Java* gibt es mittlerweile diverse Frameworks, welche den Umgang mit der Kommandozeile vereinfachen.

Es wurde sich für den Einsatz des *picocli*-Frameworks[5] entschieden, da die Konfiguration über Annotationen funktioniert und eine automatische „type-conversion“, also Umwandlung der Eingabeparameter in den entsprechenden angegebenen Typ, stattfindet. Weiterhin werden durch Setzen eines Parameters das „–help“ und „–version“ Kommando automatisch zur Verfügung gestellt. Dadurch bleibt der Code übersichtlich und selbsterklärend.

Ein Ausschnitt des Quelltextes für den `GeneratorCommand` ist in Listing 9.2 dargestellt. Hierbei wird das Zusammenspiel der Annotationen und Attribute deutlich. Ausserdem wird deutlich, wie wenig zusätzlicher Code notwendig ist, um das Programm für den parametrisierten Aufruf von der Kommandozeile vorzubereiten.

5.3.2 Eingabeparameter

Der Software können verschiedene Eingabeparameter übergeben werden. Die möglichen Optionen sind in Tabelle 5.1 dargestellt. Bis auf die Angabe der *OHDM*-Konfiguration sind alle optional.

[4] OSMImportUpdate-Projekt: https://github.com/OpenHistoricalDataMap/OSMImportUpdate
[5] Projektseite von *picocli*: https://picocli.info/

Parameter	Kommando	Beschreibung
OHDM-Konfigurationsdatei	Pflichtparameter (kein Kommando vorhanden)	Pfad zu einer Datei, die die Anweisungen für die Generierung der Styles enthält.
Datenbank-Konfigurationsdatei	„-db“ oder „–db-config“	Pfad zu einer Datei, die Parameter zur Erzeugung einer Verbindung zu einer *PostGIS* Datenbank enthält.
GeoServer-Konfigurationsdatei	„-c“ oder „–geoserver-config“	Pfad zu einer Datei, welcher Informationen über einen *GeoServer* enthält.
Ausgabeverzeichnis	„-o“ oder „–output-dir“	Pfad zu einem Verzeichnis, wo die generierten Styles abgelegt werden sollen.
Generierung von Standards	„-d“ oder „–generate-defaults“	Boolscher Wert, welcher bestimmt ob für, in der *Classification* existierende, aber in der *OHDM*-Konfiguration nicht beschriebene, Klassen Standardwerte generiert werden sollen.

Tabelle 5.1: Übersicht der möglichen Eingabeparameter

OHDM-Konfigurationsdatei

Die Angabe eines Pfades zu einer *OHDM*-Konfigurationsdatei ist obligatorisch. Der Aufbau bzw. Inhalt dieser Dateien wird in Abschnitt 4.2.2 beschrieben.

Datenbank-Konfigurationsdatei

Es ist möglich eine Datei mit Informationen zu einer *PostGIS*-Datenbank zur Verfügung zu stellen. In diesem Fall wird die Software versuchen, eine *Classification* anhand der Datenbank zu erzeugen und eventuell fehlende Tabellen automatisch erzeugen. Benötigt werden hierbei Angaben über Hostadresse, Datenbankname, Portnummer sowie Benutzername und Passwort eines Benutzers der Datenbank. Die Angabe eines Schemas ist optional, standardmässig wird das „public“-Schema verwendet. Ein Beispiel für eine Datenbank-Konfigurationsdatei ist in Listing 9.4 abgebildet.

GeoServer-Konfigurationsdatei

Um die generierten Styles automatisch auf einen *GeoServer* hochzuladen ist die Angabe einer *GeoServer*-Konfigurationsdatei notwendig. Hier müssen Angaben gemacht werden, welche zur Erstellung einer Verbindung zum Host gebraucht werden. Dazu zählen die Adresse des Hosts sowie

Benutzername und Passwort für einen Benutzer, der auf dem Server existiert. Weiterhin wird die Angabe eines Pfades zum Hauptverzeichnis der Daten (siehe Abschnitt 4.3.1) des *GeoServers* benötigt.

Unter bestimmten Umständen optional ist die Angabe von Namen für *Workspace*, *Namespace* und *Store* sowie eines Coordinate Reference System (CRS)-Codes (also eine Angabe über das zu verwendende Koordinatensystem). Dabei gibt es drei verschiedene Möglichkeiten:

- Wenn Namen für *Workspace* und *Namespace* angegeben werden, wird versucht die Styles in dem „styles“-Verzeichnis des *Workspaces* abzulegen.
- Wenn keine Namen für *Workspace* und *Namespace* angegeben werden, werden die Styles im globalen „styles“-Verzeichnis abgelegt.
- Wenn dem Programm ausserdem eine Datenbank-Konfigurationsdatei übergeben wurde, so müssen Namen für *Workspace*, *Namespace* und *Store* angegeben werden. Die Angabe eines CRS ist optional und nur dann relevant, wenn *GeoServer*-Konfigurationsdatei und Datenbank-Konfigurationsdatei zusammen verwendet werden. Als Standardwert wurde „EPSG:3857“ definiert, da dieser im *OHDM*-Projekt standardmässig verwendet wird.

Listing 9.3 zeigt eine beispielhafte Datei.

Generierung von Standards

Bei Bedarf können in der *OHDM*-Konfiguration nicht beschriebene Klassen/*Map Features* mit Standardregeln generiert werden. Dabei werden für jede Subklasse jeweils ein Style für Linien, Punkte und Polygone erzeugt, welche dafür sorgen, dass auch die nicht speziell konfigurierten *Map Features* auf einer Karte dargestellt werden.

Auswirkungen auf den Programmablauf

Der Ablauf des Programms ist (in Abbildung 5.1 dargestellt und) abhängig von den beim Start über die Kommandozeile übergebenen Argumenten. Wenn keine *OHDM*-Konfiguration existiert wird das Programm sofort beendet. Falls sie existiert wird zuerst eine `Classification` erzeugt und anschließend die *SLD*-Dateien auf Basis der *OHDM*-Konfiguration erzeugt. Sofern eine *GeoServer*-Konfigurationsdatei übergeben wurde, wird davon ausgegangen, dass die *SLD*-Dateien auf den angegebenen Server hochgeladen werden sollen. Davor wird geprüft, ob auch eine Datenbank-Konfiguration existiert und davon abhängig entschieden, in welchem Ausmaß die weiteren Konfigurationsdateien erzeugt werden können. Wenn die Datenbank-Konfiguration existiert, werden

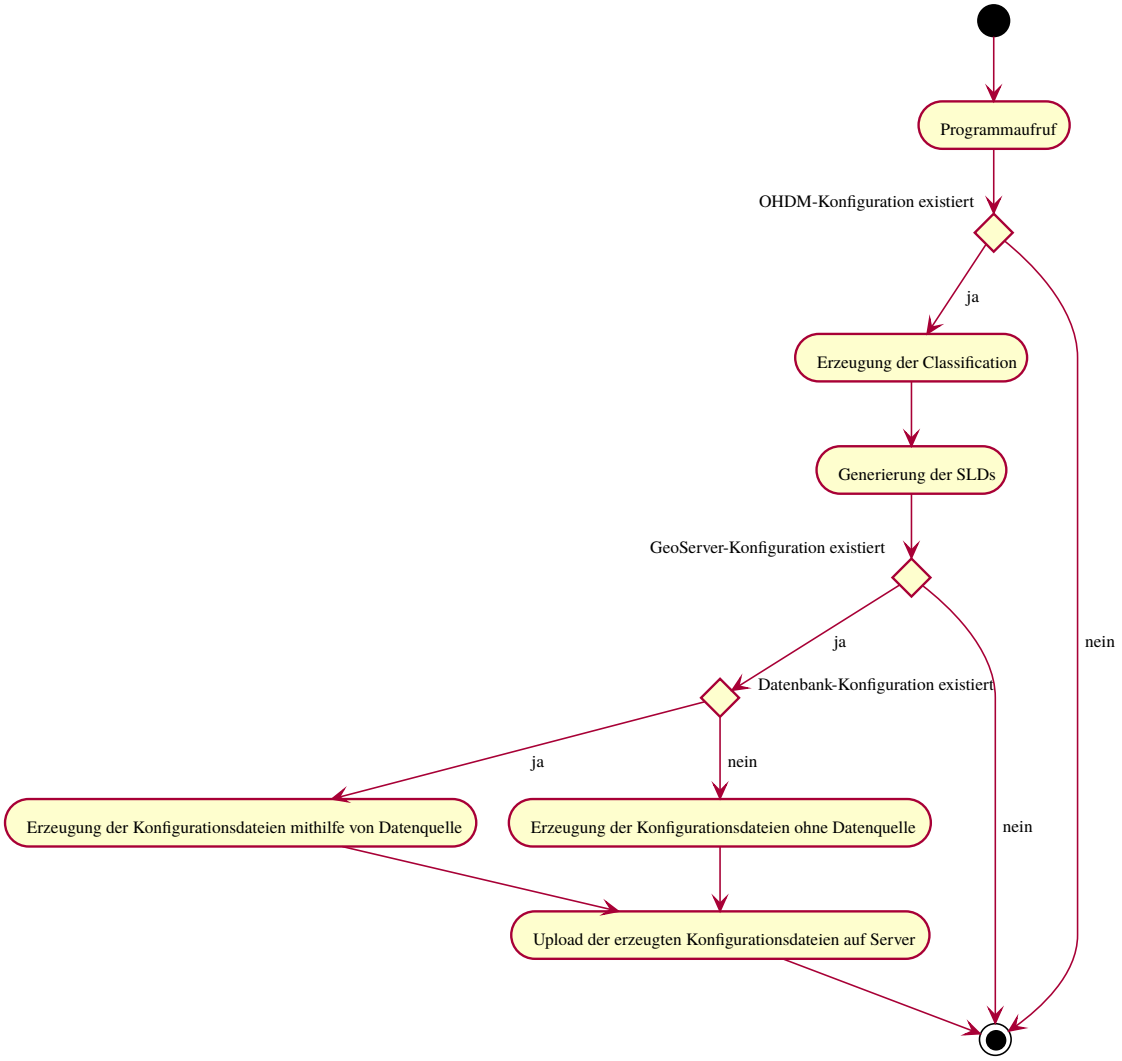

Abbildung 5.1: Darstellung des Programmablaufs in einem Aktivitätsdiagramm

mithilfe dieser weitere Einstellungen erzeugt, falls nicht werden nur die Styling-spezifischen Konfigurationsdateien erzeugt. Die im vorherigen Schritt erzeugten Dateien werden schließlich auf den angegebenen Server hochgeladen und das Programm beendet.

5.4 Implementierung des Parsers

5.4.1 Verwendetes Framework

Für das Parsen der *OHDM*-Konfigurationsdatei wird das *parboiled*-Framework[6] verwendet. *Parboiled* ist eine Java bzw. Scala Bibliothek, welche das Parsen von Text mithilfe von *Parsing Expression Grammars* ermöglicht. Abgesehen davon, dass überprüft werden kann, ob ein gegebener Input den Regeln der Grammatik entspricht, stellt *parboiled* einen „Value Stack“ zur Verfügung, welcher dazu genutzt werden kann, um Werte während des Parsens zu verarbeiten. Die Verwendung davon wird in Abschnitt 5.4.2 genauer erläutert. Weiterhin wird das Framework von der *CSS*-Extension verwendet, sodass durch Benutzung dieser Bibliothek auf die Implementierung der Regeln der *CSS*-Extension zurückgegriffen werden kann.

5.4.2 Die `ConfigParser`-Klasse

Das Parsen eines Inputs auf Basis der Regeln ist in der `ConfigParser`-Klasse implementiert. *Parboiled* bietet eine simple Syntax für die Definition der Regeln an. Diese soll anhand von Listing 5.1 erläutert werden.

Jede Regel wird durch eine eigene Methode implementiert. Dabei entspricht der Methodenname dem Namen der Regel. Die Regeln geben ein Objekt zurück, welches ihren Aufbau beschreibt.

In den Zeilen 236-242 sieht man, dass die Implementierung sehr ähnlich zu der Regel in der Definition der Grammatik (aus Abschnitt 4.2.3) ist.

Bei der Implementierung der `StylePlaceholderDeclaration`-Regel wird eine Besonderheit von *parboiled* aufgezeigt. In Zeile 251 wird ein Wert auf dem „Value Stack“ platziert. Die Methode `match()` gibt immer den Wert der letzten übereinstimmenden Regel zurück. In diesem Fall wird also der Wert der `StyleName()` Regel auf dem „Value Stack“ abgelegt. Anschließend wird in den Zeilen 252-260 eine sogenannte `Action` definiert, welche an dieser Stelle ausgeführt werden soll. Hier wird zuerst durch `pop()` der oberste Wert des „Value Stacks“ abgefragt. Es kann davon ausgegangen werden, dass der Wert an dieser Stelle immer dem `StyleName` entspricht, da er als Letztes auf den Stack gelegt wurde, oder bei Nichterfüllung der vorherigen Regeln, die `Action` gar nicht erst ausgeführt werden würde. Der Wert wird dann benutzt um eine Instanz der `PlaceholderRule`-Klasse zu erzeugen und diese anschließend auf dem Stack zu platzieren.

[6] Link zur Projektseite: https://github.com/sirthias/parboiled/wiki

```
236     Rule StyleDeclaration() {
237         return Sequence(
238                 OptionalWhiteSpace(),
239                 FirstOf(StylePlaceholderDeclaration(), DetailedStyleDeclaration()),
240                 OptionalWhiteSpace()
241         );
242     }
243 
244     Rule StylePlaceholderDeclaration() {
245         return Sequence(
246                 String("useStyle"),
247                 OptionalWhiteSpace(),
248                 EqualsSymbol(),
249                 OptionalWhiteSpace(),
250                 StyleName(),
251                 push(match()),
252                 new Action() {
253                     @Override
254                     public boolean run(Context context) {
255                         String referencingStyleGroupName = (String) pop();
256                         PlaceholderRule placeholderRule = new PlaceholderRule(
referencingStyleGroupName);
257                         push(placeholderRule);
258                         return true;
259                     }
260                 }
261         );
262     }
```

Listing 5.1: Ausschnitt der `ConfigParser`-Klasse

Dieses Beispiel zeigt das grundsätzliche Vorgehen bei der Implementierung von Regeln mit *parboiled*. Die Regeln erzeugen neue Objekte auf Basis der Informationen, welche ihnen an dieser Stelle zur Verfügung stehen, wobei die Abstraktionsebene bei den tieferen Regeln sinkt.

Das Resultat des Parsens wird in einer Instanz der Klasse `ConfigParseResult` gespeichert. Dieses Objekt ist ein simpler Wrapper mit der Aufgabe die `MapFeatures` und `StyleGroups` zu halten.

5.5 Erzeugung der SLD-Dateien

Für die Erzeugung der *SLD*-Dateien wird auf Implementierungen der „geotools“-Bibliothek[7] zurückgegriffen. Diese *Java*-Bibliothek stellt Methoden für die Verarbeitung von Geodaten zur Verfügung, welche den Standards des *OGCs* entsprechen [Ope19b]. Relevant sind in diesem Fall die

[7] Projektseite von *geotools*: https://geotools.org/

`StyledLayerDescriptor-` und `SLDTransformer`-Klasse. Die `SLDTransformer`-Klasse ist in der Lage aus einer Instanz der `StyledLayerDescriptor`-Klasse eine standardkonforme *SLD*-Datei zu erzeugen. Das Ziel ist deshalb das Ergebnis des Parsens in `StyledLayerDescriptor`-Instanzen zu überführen.

5.5.1 Verarbeitung des `ConfigParseResults`

Die Verarbeitung des `ConfigParseResults` ist Aufgabe eines `ParseResultProcessors`. Dieser wird mit einer `Classification` initialisiert und stellt eine Methode zur Verfügung, welche, bei Übergabe eines `ConfigParseResult` als Argument, eine Sammlung von `StyledLayerDescriptor`-Objekten zurückgibt. Die Funktionsweise dieser Methode ist in Abbildung 5.2 dargestellt.

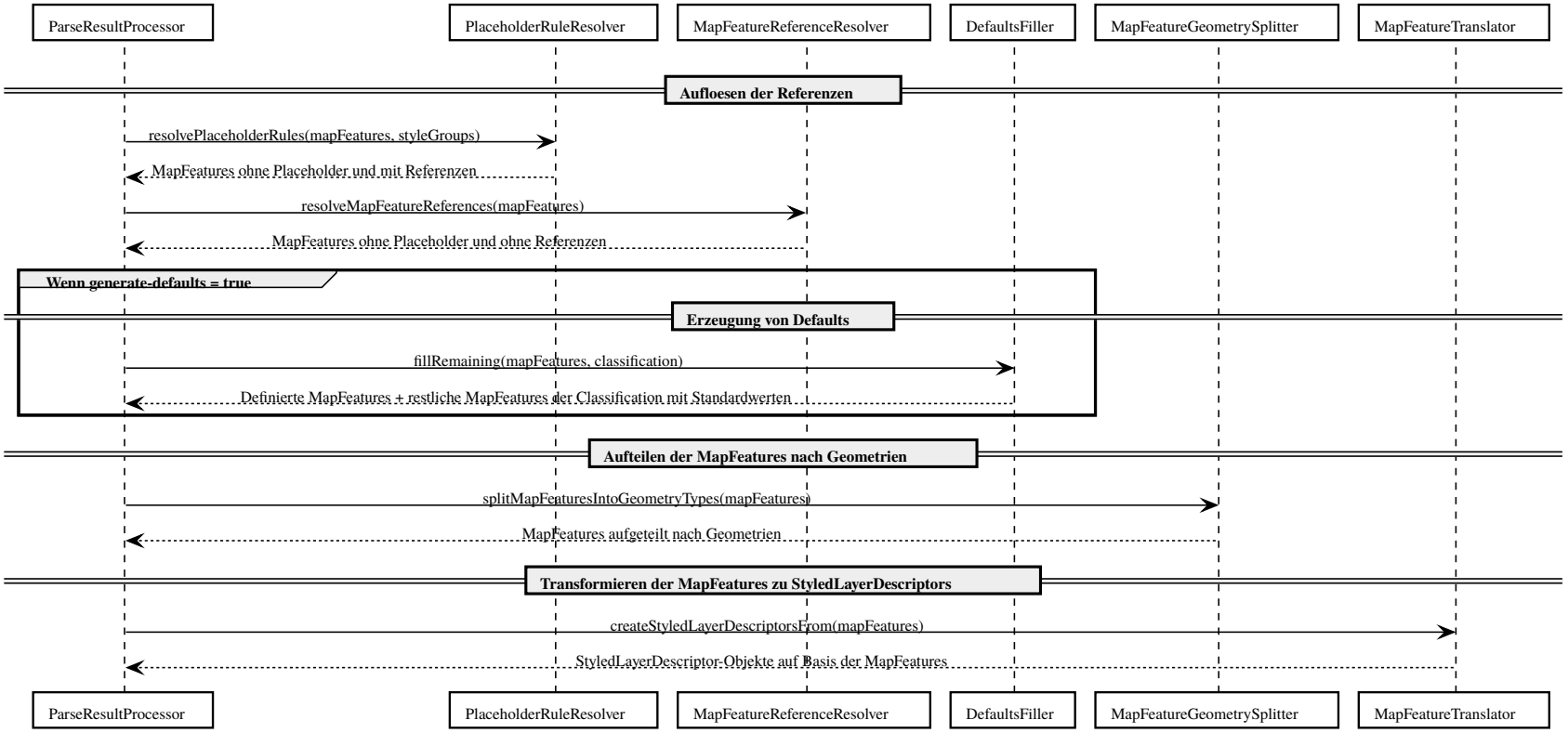

Abbildung 5.2: Ablauf der Verarbeitung des `ConfigParseResults`

Das Ziel des `ParseResultProcessors` ist die Erzeugung von `StyledLayerDescriptors` auf Basis des `ConfigParseResults`, denn mithilfe dieser Objekte können *SLD*-Dateien erzeugt werden.

Auflösen der Referenzen

Der erste Schritt ist es, die Referenzen zu `StyleGroups`, welche mit „useStyle“ gesetzt wurden, aufzulösen. Diese Referenzen sind in Form von `PlaceholderRule`-Instanzen abgebildet, welche eine Implementierung des `Rule`-Interfaces darstellen und zusätzlich den Namen der referenzierten `StyleGroup` enthalten.

Zunächst werden die Referenzen der `StyleGroups` untereinander aufgelöst und anschließend die Referenzen von `MapFeatures` zu `StyleGroups`. Dabei wird jede `PlaceholderRule` mit einer Kopie der referenzierten Regel ersetzt, welche von der ursprünglichen Regel insofern abweicht, dass die Filter beider Regeln kombiniert und die Zoom-Einstellungen des Placeholders verwendet werden.

Der zweite Schritt ist das Auflösen der Referenzen zwischen `MapFeatures`, welche mit „sameAs“ gesetzt wurden. Hierbei werden nur die Regeln von Subklassen übernommen, welche für beide `MapFeatures` existieren. Das bedeutet, dass (bei Verwendung der `OSMClassification`) in den meisten Fällen nur die Regeln für die Subklasse „undefined“ übernommen werden, da diese für fast alle `MapFeatures` existiert.

Erzeugung von Defaults

Falls der Nutzer den „generate-defaults“ Eingabeparameter gesetzt hat, wird die Sammlung der `MapFeatures` um die, in der *Classification* definierten, aber nicht in der *OHDM*-Konfigurationsdatei beschriebenen, `MapFeatures` erweitert. Diese erhalten einen Standardstil, sodass sie auch ohne zusätzliche Konfiguration des Nutzers Geodaten auf einer Karte darstellen. Der Standardstil ist dabei so gewählt, dass Punkte mit einem schwarzen Kreis, Linien mit einem schwarzen Strich und Polygone mit einer schwarzen Füllung dargestellt werden.

Aufteilen der `MapFeatures` nach Geometrien

Da es zu unerwünschten Nebeneffekten führt, wenn die Art der Symbolizer nicht zu der Geometrie passt, auf welche sie angewendet werden (siehe Abschnitt 5.6.1), müssen die `MapFeatures` aufgeteilt werden. Das bedeutet konkret, dass die Regeln eines `MapFeatures` auf andere (neue) `MapFeature`-Instanzen aufgeteilt werden, welche dann für jeweils einen Geometrietyp zuständig sind.

Für ein besseres Verständnis wird der Ablauf im Folgenden beispielhaft beschrieben: Es wird angenommen, dass ein `MapFeature` für die Klasse „boundary“ existiert. Deshalb werden auf Basis des „boundary“-`MapFeatures` die `MapFeatures` „boundary_lines“, „boundary_polygons“ und „boundary_points“ erzeugt. Folglich werden „boundary_lines“ die Regeln mit einem `LineSymbolizer`, „boundary_polygons“ die Regeln mit einem `PolygonSymbolizer` und „boundary_points“ die Regeln mit einem `PointSymbolizer` zugewiesen. `TextSymbolizer` sind hiervon ausgeschlossen und bleiben bei allen Regeln erhalten, da ihre Darstellung unabhängig vom Geometrietyp ist.

Transformieren der `MapFeatures` zu `StyledLayerDescriptors`

Schlussendlich werden die zuvor entstandenen `MapFeatures` zu Instanzen von `StyledLayerDescriptors` transformiert. Der Name wird übernommen und die Informationen zu Subklassen und Styling an den entsprechenden Stellen der neuen Struktur platziert.

5.5.2 Speichern der SLD-Dateien

Der `SLDTransformer` ist für das Übersetzen der `StyledLayerDescriptors` in valide *SLD*-Beschreibungen zuständig. Dafür benötigt er die Instanz des Objektes und den Pfad zur anzulegenden Datei. Als Zielverzeichnis wird entweder das vom Nutzer angegebene oder das Standardausgabeverzeichnis gewählt. Der Dateiname wird äquivalent zum Namen des `StyledLayerDescriptors` gewählt (und erhält die Endung „.sld“).

5.6 Konfiguration der Datenquelle

5.6.1 Problematik

Wenn man für Geodaten, deren Geometrietypen sich unterscheiden, unterschiedliche Styles definieren will, gibt es einige Besonderheiten zu beachten. Da die verschiedenen Symbolizer im *SLD 1.0*-Standard auf alle Geometrietypen angewendet werden können, müssen zusätzliche Vorkehrungen getroffen werden, um sicherzustellen, dass die Symbolizer nur auf ihren entsprechenden Geometrietyp angewandt werden (vgl. [Ope19i]).

Im *OHDM*-Projekt wurde sich dafür entschieden, die Tabelle jedes *Map Features* aufzuteilen, sodass jede Tabelle Daten für einen anderen Geometrietyp beinhaltet. Es wurden also für jedes *Map Feature* drei Tabellen erzeugt: eine für Linien, eine für Punkte und eine für Polygone.

5.6.2 Erzeugung der Tabellen

Die Software wird versuchen für jede in der *Classification* vorhandene Klasse Tabellen zu erzeugen, sofern diese noch nicht existieren. Bei der Erzeugung der Tabellen wird die Vorgehensweise aus Abschnitt 5.6.1 verwendet. Damit wird sichergestellt, dass bei der anschließenden Konfiguration des *GeoServers* die *FeatureTypes*, welche auf diesen Tabellen basieren, konstruiert und konfiguriert werden können. Das bedeutet allerdings, dass die *Layer* sobald die Tabellen mit Geodaten gefüllt werden, manuell konfiguriert werden müssen, da die *Bounding Boxes*, welche anhand der in der Tabelle vorhandenen Geodaten berechnet werden, nicht mehr gelten.

5.7 Architektur der Konfigurationssoftware

Um die Dateien, welche für die automatisierte Konfiguration des *GeoServers* nötig sind, zu erzeugen wird auf die eigentliche Implementierung des *GeoServers* zurück gegriffen. Alternativ könnte man auch dazu übergehen, existierende Dateien als Basis zu nehmen und diese versuchen im Code nachzubauen. Durch die Verwendung der *GeoServer*-Implementierung kann aber sichergestellt werden, dass bei etwaigen Veränderungen der Datenstrukturen in späteren Versionen keine aufwendigen Änderungen des Codes erforderlich sind, sodass leichter auf neuere Versionen aktualisiert werden kann.

5.7.1 GeoServer-Module

Die Software des *GeoServer*-Projektes ist Open-Source und wird auf GitHub gehostet[8]. Die Funktionalitäten des *GeoServer*-Projektes sind in zahlreiche Untermodule unterteilt wobei Maven-Artefakte der einzelnen Teilmodule von `boundlessgeo.com` zur Verfügung gestellt werden. In diesem Projekt werden drei dieser Teilmodule verwendet.

Verwendete Version

Die aktuellste stabile Version des *GeoServers* zum Zeitpunkt der Entstehung dieses Projektes ist Version 2.15.2. Obwohl im *OHDM*-Projekt derzeit eine ältere Version des *GeoServers* eingesetzt wird, wurde in dieser Arbeit die aktuellste Version verwendet, da es sinnvoller ist, die Software im Rahmen dieser Arbeit auf dem neuesten Stand zu entwickeln und den *GeoServer* des *OHDM*-Projektes auf eine neuere Version zu updaten. Die im nachfolgenden besprochenen Module basieren alle auf dieser Version.

[8] GitHub-Seite des *GeoServer*-Projekts: `https://github.com/geoserver/geoserver`

Das *Main*-Modul

Im *Main*-Modul existieren verschiedenste Klassen, die für den einfachen Betrieb einer *GeoServer*-Instanz gebraucht werden.

Die Wurzel des Gerüsts der *GeoServer*-Implementierung ist die `GeoServer`-Klasse. Diese bietet Zugriff auf Einstellungen, Metadaten, einen `Catalog`, Services und mehr.

Das Kernstück der Datenhaltung der *GeoServer*-Implementierung ist der sogenannte `Catalog`. Dieses Objekt ist für das Management der Objekte zuständig. Der `Catalog` hat Zugriff auf sämtliche Datenobjekte, welche für den *GeoServer* relevant sind und kann diese erzeugen, validieren und zurückgeben.

Weiterhin besitzt das *Main*-Modul des *GeoServers* für alle Objekte, die in Tabelle 2.1 beschrieben werden, eigene Implementierungen. Die Namen dieser Klassen enden generell mit der Endung „-Info“.

Das Zusammenspiel bzw. die Beziehungen der Objekte untereinander sind in Abbildung 5.3 dargestellt. Einige irrelevante Eigenschaften wurden zur Vereinfachung weggelassen. Aus der Abbildung wird deutlich, dass alle Assoziationen vom Typ einer Komposition sind und eine definierbare Hierarchie der Objekte existiert.

Weitere Module

Weiterhin existieren ein *OGC Web Services (OWS)*-Modul und ein *Platform*-Modul für das *GeoServer*-Projekt, welche zur ordnungsgemäßen Ausführung der Software nötig sind. Der *GeoServer* benutzt zur Erzeugung und Validierung neuer Instanzen Implementierungen aus beiden Modulen.

5.7.2 Generierung der Konfigurationsdateien

Für die Generierung der Konfigurationsdateien wird davon ausgegangen, dass eine *GeoServer*-Konfigurationsdatei als Eingabeparameter übergeben wurde. Ohne diese Datei kann eine Konfiguration nicht durchgeführt werden.

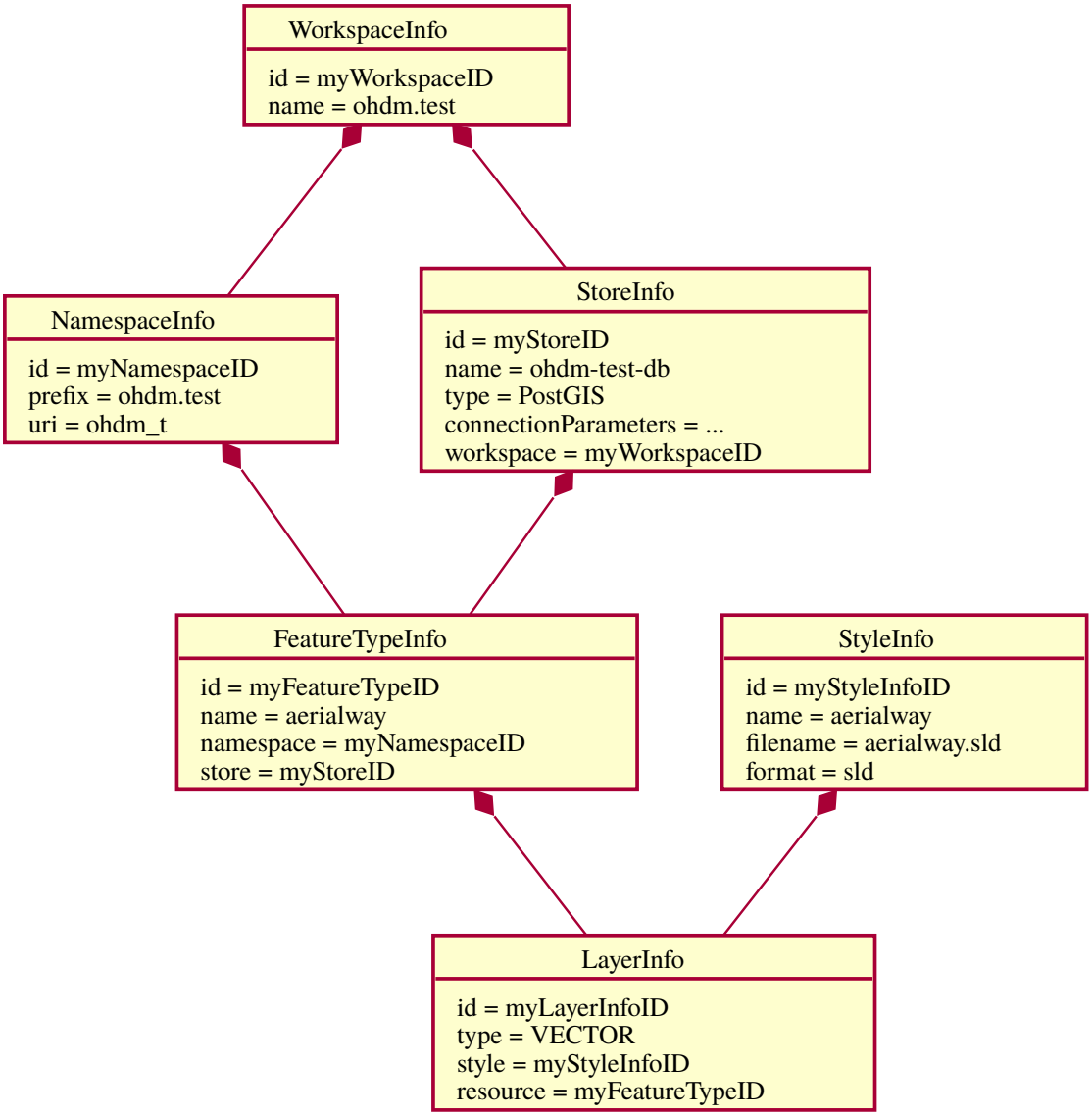

Abbildung 5.3: Darstellung der Assoziationen der Objekte in einem Objektdiagramm

Schaffen der Rahmenbedingungen

Um die Generierung der Konfigurationsdateien zu vereinfachen sollen Implementierungen der *GeoServer*-Module verwendet werden. Das eigentliche Ziel ist die Konstruktion von Instanzen der `CatalogInfo`-Objekte, welche dann in *XML*-Dateien übertragen werden können.

Damit die Erzeugung dieser Objekte gelingt müssen allerdings bestimmte Rahmenbedingungen geschaffen werden. Hiermit ist gemeint, dass das grobe Gerüst eines *GeoServers* initialisiert sein muss, bevor auf den Teil mit der Datenerzeugung zugegriffen werden kann. Dazu werden zuerst

eine Instanz der `GeoServer`- und der `Catalog`-Klasse erzeugt und miteinander verknüpft. Ohne eine Instanz der `Catalog`-Klasse können die meisten Objekte nicht konstruiert werden, da die Validierung sonst fehlschlägt.

Um die Konstruktion valider Instanzen der verschiedenen `CatalogInfo`-Klassen zu erleichtern, wurde eine `CatalogInfoFactory`-Klasse implementiert, welche Methoden zur Verfügung stellt, um sämtliche relevante `CatalogInfo`-Objekte zu erzeugen. Dazu muss der `CatalogInfoFactory` die Instanz eines `Catalogs` übergeben werden. Die Factory selbst benutzt Wrapper-Klassen, die für jede relevante „*-Info“-Klasse implementiert wurden und welche die obligatorischen Parameter im Konstruktor vorschreiben, sodass das Auftreten von Fehlern zur Laufzeit aufgrund fehlender Attribute verhindert wird. Die Wrapper-Klassen konfigurieren dann schlussendlich die Instanzen anhand ihrer übergebenen Parameter.

Generierung der Konfigurationsobjekte

Welche Objekte letztendlich generiert werden hängt von den an das Programm übergebenen Parametern (siehe Tabelle 5.1) ab.

Wenn eine valide Datenbank-Konfiguration existiert werden sämtliche unterstützte Konfigurationsobjekte erzeugt. Dabei werden mithilfe der *GeoServer*-Konfiguration die Namen für `Workspace`, `Namespace` und `Store` bestimmt und die Verbindungsparameter des `StoreInfo`-Objekts basierend auf den Informationen der Datenbank-Konfiguration gewählt. Für jede in Abschnitt 5.5 erzeugte *SLD*-Datei werden `StyleInfo`, `FeatureTypeInfo` und `LayerInfo` generiert.

Wenn keine Angabe einer Datenbank-Konfiguration existiert werden nur `StyleInfo`-Objekte erzeugt und die Konfiguration des `Stores` fällt weg.

Speichern der Konfigurationsobjekte

Für das Speichern der Objekte wurde eine `CatalogInfoPersister`-Klasse implementiert. Diese Klasse wird mit einem `XStreamPersister`[9] und der Angabe eines Zielverzeichnisses konstruiert und bietet für jede `CatalogInfo`-Klasse eine Methode, um diese zu speichern. Die Besonderheit dieser Methoden ist, dass die Objekte abhängig von ihrem Typ in unterschiedlichen Unterverzeichnissen des Zielverzeichnisses platziert werden. Dabei wird die Struktur des *GeoServers* nachgebildet, was für den Upload der Dateien (siehe Abschnitt 5.8.3) wichtig ist.

[9] Utility-Klasse des *GeoServers*, welche Objekte in XML-Dateien transformieren kann.

5.8 Upload der Dateien

5.8.1 Verwendetes Framework

Für die Herstellung einer Verbindung zum *GeoServer* wird das „sshj“-Framework[10] verwendet. Dieses Framework ermöglicht Kommunikation zwischen Computern via ssh, scp und sftp.

5.8.2 Voraussetzungen

Damit *sshj* unter Windows funktioniert muss „bouncycastle“ zur Verfügung stehen. *BouncyCastle* bezeichnet eine Sammlung von in *Java* implementierten kryptographischen Algorithmen (vgl. [Leg13]). Die aktuellste Version kann z. B. bei *mvnrepository.com* in Form einer *JAR*-Datei[11] bezogen werden. Diese Datei muss anschließend im `lib/ext/`-Unterverzeichnis der *Java Runtime Environment*-Installation platziert werden, damit sie während der Programmausführung zur Verfügung steht.

5.8.3 Platzierung der Dateien

Die erzeugten Konfigurationsdateien werden bei der Generierung so in einem lokalen Verzeichnis abgelegt, dass ihre Position in den Unterverzeichnissen die für den *GeoServer* erforderliche Struktur wiederspiegelt. Die relativen Pfade der lokalen Dateien zum Root-Verzeichnis sind also am Ende äquivalent zu den relativen Pfaden der auf dem *GeoServer* platzierten Dateien zum Hauptverzeichnis der Daten der *GeoServer*-Installation.

Hierbei ist anzumerken, dass aufgrund der Tatsache, dass die Konfigurationsdateien lokal erzeugt werden, ohne sie mit den bereits vorhandenen Konfigurationen des *GeoServers* abzugleichen, die vorhandenen Einstellungen eines *Workspaces* überschrieben werden, sofern dieser bereits mit gleichem Namen existiert. Das hat zur Folge, dass sich die IDs der dort abgelegten *GeoServer*-Objekte ändern, auch wenn die Informationen gleich bleiben. Problematisch wird es dann, wenn aus anderen *Workspaces* auf Styles in dem neu konfigurierten *Workspace* referenziert wird, da die IDs nicht mehr übereinstimmen. Da die Styles eines *Workspaces* in der Regel nur in selbigem benutzt und referenziert werden, sollte es nur selten zu Problemen kommen.

[10] GitHub-Seite des Projekts: https://github.com/hierynomus/sshj
[11] Dabei sollte die Dependency „bcprov-jdk15on“ gewählt werden

5.9 Logging

Im folgenden Abschnitt wird die Implementierung des Loggings erläutert. Die Notwendigkeit dafür geht aus den Benutzbarkeitsanforderungen aus Abschnitt 3.2.3 hervor.

5.9.1 Framework

In diesem Projekt wird das Logging-Framework „Apache Log4j 2“[12] eingesetzt. Dieses von der *Apache Software Foundation* entwickelte Framework stellt zahlreiche Konfigurationsmöglichkeiten bereit, um die Logging-Ausgaben an die Anforderungen des Projektes anzupassen. Weiterhin bietet es eine Vielzahl von hilfreichen Methoden, um Entwicklern das Verfassen von Logging-Statements zu erleichtern.

Eine detaillierte Übersicht mit Vorteilen von *Log4j* findet man unter https://logging.apache.org/log4j/2.x/manual/index.html.

5.9.2 Konfiguration

Das Framework kann entweder programmatisch oder über Konfigurationsdateien konfiguriert werden. Da sich die Konfiguration zur Laufzeit nicht ändern wird, wurde sich für die Benutzung einer Datei entschieden. Diese wurde im *XML*-Format geschrieben und sorgt dafür, dass die Log-Statements in einem bestimmten Format (konkret: „[Log-Level] Loggende Klasse: Log-Nachricht“) auf der Konsole ausgegeben werden. Auf die Erzeugung einer separaten Log-Datei wurde verzichtet, da das Programm prinzipiell immer über die Konsole aufgerufen wird, weshalb der Nutzer sowieso sofortige Einsicht in die Log-Nachrichten erhält.

5.9.3 Protokollierte Informationen

Das Programm informiert den Nutzer über den aktuellen Stand der Verarbeitung. Das bedeutet, dass vor jeder wichtigen Aktion eine Information ausgegeben wird, welche Aktion als Nächstes gestartet wird. Dadurch lässt sich im Fehlerfall nachvollziehen, an welcher Stelle der Fehler auftrat, wodurch die möglichen Fehlerquellen eingeschränkt und dadurch die Fehlerbehebung vereinfacht werden kann. Weiterhin wird bei Speichern eines Objektes in eine Datei der Pfad ausgegeben, damit der Nutzer nachvollziehen kann, wo genau die Dateien abgelegt wurden.

[12] Projektseite von *Log4j*: https://logging.apache.org/log4j/2.x/index.html

6 Tests

6.1 Einrichtung der Testumgebung

Um die Software umfassend testen zu können werden zusätzliche Komponenten benötigt. Dazu gehören eine laufende Instanz eines *GeoServers* und eine Datenbank.

6.1.1 Einrichtung der Datenbank

Auf die Einrichtung einer eigenen *PostGIS*-Datenbank wurde verzichtet. Die bloße Installation einer solchen ist zwar trivial, da allerdings das Vorhandensein von Geodaten erforderlich ist, um den Einsatz der Software zu überprüfen, wurde auf die im *OHDM*-Projekt eingesetzte *PostGIS*-Datenbank zurückgegriffen.

6.1.2 Einrichtung eines virtuellen *GeoServers*

Der im *OHDM*-Projekt momentan verwendete *GeoServer* basiert auf Version 2.7.6. Da die im Rahmen dieser Arbeit entwickelte Software auf der aktuellsten Version des *GeoServers* ausführbar sein soll, wurde eine virtuelle Umgebung aufgesetzt, in welcher die aktuelle Version (2.15.2) der *GeoServer*-Software installiert wurde.

Die detaillierte Anleitung zur Installation des *GeoServers* ist in der offiziellen Dokumentation[1] beschrieben.

Die Einrichtung der Testumgebung auf einer virtuellen Maschine mit der Distribution `Ubuntu Server 18.04` als Betriebssystem wird im Folgenden kurz zusammengefasst.

1. Java 8 Runtime Environment (JRE) installieren (`sudo apt install openjdk-8-jdk`)
2. „openssh-server“ installieren, um SSH-Verbindungen zu ermöglichen (`sudo apt install openssh-server`)

[1] Link zur Installationsanleitung https://docs.geoserver.org/stable/en/user/installation/index.html

3. *GeoServer*-Binary herunterladen (`wget -O geoserver-binary.zip "https://downloads.sourceforge.net/project/geoserver/GeoServer/2.15.2/geoserver-2.15.2-bin.zip?r=https%3A%2F%2Fsourceforge.net%2Fprojects%2Fgeoserver%2Ffiles%2FGeoServer%2F2.15.2%2Fgeoserver-2.15.2-bin.zip%2Fdownload&ts=1564927261"`)
4. Zielverzeichnis der Installation erzeugen (`mkdir /usr/share/geoserver`)
5. Zip-Datei entpacken (`unzip geoserver-binary.zip`)
6. Inhalt der Zip-Datei zu Zielverzeichnis verschieben (`mv geoserver-2.15.2/* /usr/share/geoserver`)
7. Umgebungsvariable setzen (`echo "export GEOSERVER_HOME=/usr/share/geoserver" >> ~/.profile && . ~/.profile`)
8. Besitzer wechseln (`sudo chown -R USER_NAME /usr/share/geoserver/`)
9. *GeoServer* starten (`/usr/share/geoserver/bin/startup.sh`)
10. Auf *Web Administration Interface* wechseln (`http://lokale_ip_der_vm:8080/geoserver/web`)
11. Einloggen mit Benutzername „admin“ und Passwort „geoserver“

Hinweis Wenn als *Store* die *PostGIS*-Datenbank des *OHDM*-Projektes verwendet werden soll und man sich nicht im Netzwerk der HTW-Berlin befindet, muss in der virtuellen Umgebung eine *Virtual Private Network (VPN)*-Verbindung zur HTW Berlin hergestellt werden, da die Datenbank nur innerhalb des Netzwerks der Hochschule erreichbar ist.

6.2 Testarten

In diesem Abschnitt soll auf, für das Testen dieses Systems geeignete, Testarten und deren Implementierung eingegangen werden. Die programmierten Tests wurden unter Verwendung des *JUnit*-Frameworks[2] umgesetzt.

[2]Framework zur Erstellung von automatisierten Unittests in Java

6.2.1 Unittests

Um die Funktionsfähigkeit der einzelnen Komponenten in Isolation zu testen, wurden Unittests implementiert. Dabei wurde auf die Anfertigung von Tests für trivialere Komponenten verzichtet und sich auf die besonders kritischen Bereiche fokussiert. Infolgedessen wurden keine Tests für die `Model-` und `POJO`-Klassen angefertigt.

Zusätzlich wurde auf die Erstellung von Negativ-Testfällen geachtet, um sicherzustellen, dass die Software beim Auftreten bestimmter Fehlerkonstellationen entsprechend reagiert.

6.2.2 Integrationstests

Es wurde ein Integrationstest implementiert, welcher die Ausführung des Programmes auf der Kommandozeile testet. Dort werden die verschiedenen möglichen Kombinationen der Eingabeparameter getestet. Dieser Test kann vom Programmierer genutzt werden, um festzustellen, ob das Programm ordnungsgemäß funktioniert. Um diesen Test durchzuführen, müssen vorher in dem „test/resources/.env“-Verzeichnis eine `geoserver-config.json` (entspricht einer *GeoServer*-Konfigurationsdatei aus Abschnitt 5.3.2) und eine `datasource-config.json` (entspricht einer Datenbank-Konfigurationsdatei aus Abschnitt 5.3.2) angelegt und mit Informationen über die zu verwendende Testumgebung befüllt werden.

Aufgrund der Tatsache, dass die Verarbeitung der Styles Aufgabe des *GeoServers* ist, wird es schwierig automatisiert zu testen, ob die erzeugten Darstellungen der Geodaten nach erfolgreichem Durchlaufen des Programms den Erwartungen entsprechen. Deshalb ist es momentan notwendig über manuelle Abfragen die Richtigkeit der Darstellungen zu überprüfen.

6.2.3 User- und Usability-Tests

Auf die Durchführung von User- und Usability-Tests wurde aufgrund des kurzen Entwicklungszeitraumes verzichtet. Entsprechende Tests sind nachgeschaltet durchzuführen, um zu evaluieren, welche Features optimiert bzw. verändert werden müssen.

6.2.4 Testen auf unterschiedlichen Plattformen

Um sicherzustellen, dass die entwickelte Software plattformunabhängig funktioniert, wurde die Ausführung auf einem Linux- und einem Windows-System getestet. Da der kritische Aspekt hauptsächlich die unterschiedliche Schreibweise der Pfade ist (Schrägstrich und umgekehrter Schrägstrich), wurde auf das zusätzliche Testen in einer macOS-Umgebung verzichtet, da Linux und macOS in diesem Aspekt kongruieren.

6.3 Testdurchführung

Der Integrationstest wurde auf die eingerichtete Testumgebung (aus Abschnitt 6.1) sowie auf den aktuell im *OHDM*-Projekt eingesetzten *GeoServer* angewendet. Dabei wurde manuell überprüft, ob die Konfiguration erwartungsgemäß abgeschlossen wurde. Das bedeutet, dass mithilfe des *Web Administration Interfaces* des *GeoServers* untersucht wurde, ob die Styles mit denen in der, für den Test eingesetzten, *OHDM*-Konfigurationsdatei übereinstimmen. Weiterhin wurde kontrolliert, dass die Einrichtung des definierten *Workspaces* sowie die Zuordnung der Styles zu den entsprechenden *Layern* erwartungsgemäß durchgeführt wurde. Dieser Test wurde auf einem Linux- und einem Windows-System ausgeführt, um die plattformunabhängige Funktionsfähigkeit sicherzustellen.

7 Bewertung der Ergebnisse

In diesem Kapitel soll die im Rahmen dieser Arbeit entwickelte Software anhand der Erfüllung der in Kapitel 3 aufgestellten Anforderungen bewertet werden. Die Ergebnisse des Testkapitels werden dabei zusätzlich betrachtet.

7.1 Erfüllung der funktionalen Anforderungen

In welchem Ausmaße die funktionalen Anforderungen realisiert wurden wird im Folgenden zusammengefasst.

7.1.1 Eingabeparameter

Alle in den Anforderungen aufgeführten Eingabeparameter wurden implementiert und erfüllen bei Betrachtung der durchgeführten Unittests nachweislich ihre entsprechende Funktion. Damit sind die funktionalen Anforderungen an die Eingabeparameter als erfüllt zu betrachten.

7.1.2 OHDM-Konfigurationsdatei

Mithilfe des Entwurfes des eigenen Dateiformates konnten die Anforderungen bezüglich Definition und Verlinkung von Klassen-Gruppierung erfüllt werden. Weiterhin wurde eine sinnvolle Einteilung in Zoomstufen, basierend auf der Empfehlung der *OSM*-Community (siehe Abschnitt 2.2.2), vorgenommen. Darüber hinaus wurde die fakultative Anforderung hinsichtlich der separaten Definition von Styles ausserhalb von Klassen-Gruppierungen und deren Referenzierung berücksichtigt und implementiert.

7.1.3 Automatisierte Konfiguration

Anhand des Integrationstests wurde gezeigt, dass die automatisierte Konfiguration bei Vorhandensein von *GeoServer*- und Datenbank-Konfigurationsdatei möglich ist und erwartungsgemäß funktioniert. Die generierten Konfigurationsdateien werden in einem temporären Verzeichnis abgelegt und anschließend auf dem angegebenen Server platziert.

7.2 Erfüllung der nichtfunktionalen Anforderungen

Inwieweit die verschiedenen Arten nichtfunktionaler Anforderungen erfüllt wurden, wird hier zusammengefasst.

7.2.1 Zuverlässigkeitsanforderungen

Die Berücksichtigung der Negativ-Testfälle gewährleistet, dass zumindest in den meisten Szenarien ausreichend detaillierte Informationen über ein Problem an den Nutzer ausgegeben werden. Es ist allerdings schwierig jegliche Fehlerquellen abzudecken, sodass eine vollständige Abdeckung nur bedingt erreicht werden kann.

Um das Risiko einer Datenzerstörung im Fehlerfall zu minimieren wurde sich darauf beschränkt, Dateien nur zu kopieren oder zu überschreiben und nicht zu löschen. Wenn der Nutzer beispielsweise einen falschen Pfad in der *GeoServer*-Konfigurationsdatei angibt, werden letztendlich nur Dateien an falschen Orten platziert, aber es tritt kein Datenverlust ein.

Die fakultative Anforderung, dass ein Nutzer Mitteilungen mit Lösungsvorschlägen zu einem konkreten Problem erhält, wurde noch nicht implementiert. Dies kann zu einem späteren Zeitpunkt unter Berücksichtigung von User-Tests noch ergänzt werden.

7.2.2 Umgebungsanforderungen

Durch die Entwicklung der Software in *Java* wird gewährleistet, dass das Programm plattformunabhängig ausführbar ist. Dies wurde durch den Test in Abschnitt 6.2.4 zusätzlich nachgewiesen.

Die Tatsache, dass der implementierte Integrationstest auf die aktuellste sowie eine älteren Version des *GeoServers* angewendet wurde, und in beiden Fällen durch manuelle Überprüfung sichergestellt werden konnte, dass die Konfiguration erwartungsgemäß vollendet wurde, zeigt, dass die Software unabhängig von der eingesetzten *GeoServer*-Version funktioniert.

7.2.3 Benutzbarkeitsanforderungen

Ob die Syntax des eingesetzten Dateiformates intuitiv und leicht zu erlernen ist, ließe sich am besten mithilfe von User- und Usability-Tests feststellen, welche allerdings, wie in Abschnitt 6.2.3 erwähnt, im Rahmen dieser Arbeit nicht durchgeführt wurden.

Die verbleibenden Anforderungen wurden allesamt erfüllt: Der Nutzer kann die vorhandenen Optionen von der Software erfragen, das Dateiformat ist für Menschen lesbar, eine automatisierte Konfiguration des *GeoServers* ist möglich und ein Nutzer erhält textuelle Informationen über den Ablauf des Programms in Form von Logging-Statements.

7.3 Auswertung

Sowohl die funktionalen als auch die nichtfunktionalen Anforderungen wurden ganzheitlich berücksichtigt und implementiert. Die Software funktioniert plattformunabhängig und ist flexibel bezüglich der Anwendung auf unterschiedliche *GeoServer*-Versionen. Durch die Tests wurde sichergestellt, dass der Betrieb der Software zuverlässig und risikofrei für den Nutzer ist. Ausschließlich die Anforderung bezüglich der Benutzbarkeit des Dateiformates lässt sich nicht hinreichend beurteilen.

8 Zusammenfassung

Das Ziel dieser Arbeit war die Erstellung einer Software, welche Dateien mit Styling-Anweisungen für einen *GeoServer* generiert. Durch eine automatisierte Konfiguration mithilfe der Verarbeitung weniger Konfigurationsdateien sollte es ermöglicht werden, eine manuelle Bearbeitung unzähliger verschiedener Style-Dateien zu vermeiden. Ferner sollte der gesamte Konfigurationsprozess bei Änderungen der Styles eines *GeoServers* weitestgehend automatisiert werden.

Nach Erhebung und Analyse der Anforderungen wurde darauf basierend eine Software entworfen, die ein individualisiertes Dateiformat verarbeiten und ausgehend davon entsprechende Styles ableiten und in *Styled Layer Descriptors* übersetzen kann. Gleichzeitig wird durch Benutzung von Standardwerten für, in der Konfigurationsdatei nicht definierte, Elemente der Konfigurationsaufwand für einen Nutzer reduziert.

Darüber hinaus kann die repetitive manuelle Konfiguration des *GeoServers* bei Angabe zusätzlicher Informationen zu Datenquelle und Server vermieden und von der Software übernommen werden.

Die durchgeführten Tests zeigen, dass die obligatorischen an das System gestellten Anforderungen erfüllt wurden.

8.1 Schlussfolgerungen

Aus der Erfüllung der Anforderungen und der Tatsache, dass die Funktionsweise der Software mit der aktuell im *OHDM*-Projekt eingesetzten Installation des *GeoServers* getestet wurde, lässt sich schlussfolgern, dass das Ergebnis für den praktischen Einsatz im Projekt geeignet ist.

Der Einsatz im laufenden Betrieb wird zeigen, ob eine Verbesserung der Anwenderfreundlichkeit durch Implementierung weiterer Features erzielt werden kann.

8.2 Limitationen

Die im Rahmen dieser Arbeit entwickelte Software ist stark auf das *OHDM*-Projekt bezogen. Der Fokus lag daher darauf, die Anforderungen, die sich aus der Verwendung im Zusammenhang mit dem Projekt ergeben, zu erfüllen. Nichtsdestotrotz ist an mehreren Stellen eine höhere Generizität möglich. Beispielsweise ist es nicht erforderlich, dass die Geometrietypen (wie in Abschnitt 5.6.2 beschrieben) in unterschiedliche Tabellen aufgeteilt werden. Andere Szenarien werden jedoch nicht unterstützt, da sie im *OHDM*-Projekt keine Anwendung finden.

8.3 Ausblick

Unabhängig davon, dass die Konfiguration des *GeoServers* stark auf das *OHDM*-Projekt bezogen ist, lässt sich die entwickelte Software auch in anderen Projekten benutzen, um *Styled Layer Descriptors* zu generieren. Wenn man die Generizität des Systems insofern erhöht, dass auch die Konfiguration von *GeoServern* ohne Verwendung einer *PostGIS*-Datenbank bzw. eine Konfiguration mit detaillierteren Einstellungsmöglichkeiten, sodass Eigenheiten von anderen Projekten unterstützt werden, möglich wird, so kann das entwickelte System auch projektübergreifende Anwendung finden. Für die Entwicklung dieser Erweiterung wäre es allerdings erforderlich, anhand von weiteren reellen Projekten die Anforderungen detailliert zu definieren, um die Implementierung dieser Erweiterungen problemorientiert zu realisieren.

9 Anhang

9.1 Hinweise zur Benutzung der Software

In diesem Abschnitt soll die Benutzung der Software anhand von Beispielszenarien veranschaulicht werden. Die Beispiele sind auf den Einsatz im *OHDM*-Projekt bezogen.

9.1.1 Erstellen der OHDM-Konfigurationsdatei

Voraussetzung für alle nachfolgenden Szenarien ist die Existenz einer validen *OHDM*-Konfigurationsdatei, welche die gewünschten Styles enthält. Als Grundlage kann beispielsweise die in Listing 9.1 dargestellte Konfiguration benutzt werden. Für die Datei kann ein beliebiger Name gewählt werden. Im Folgenden wird davon ausgegangen, dass die Datei mit „my-styles.ohdmconfig“ benannt wurde und sich im selben Verzeichnis wie das Artefakt der Software befindet.

9.1.2 Erstellen von Styles ohne zusätzliche Konfiguration

Um Styles zu erzeugen ohne zusätzliche Konfigurationsmöglichkeiten zu benutzen, muss das Artefakt der Software mit der Konfigurationsdatei als Parameter aufgerufen werden. Die Angabe eines Ausgabeverzeichnisses ist optional. Der Aufruf des Programms sieht dann wie folgt aus:

```
$ java -jar ohdm-style-generator-1.0.jar -o styles my-styles.ohdmconfig
```

Die generierten Styles befinden sich anschließend im „styles“-Verzeichnis.

9.1.3 Erstellen von Styles mit aufgefüllten Standardwerten und ohne Upload

Im Folgenden wird beschrieben, wie zusätzliche Styles für Klassen, welche nicht in der übergebenen *OHDM*-Konfigurationsdatei definiert wurden, erzeugt werden können.

Benutzung einer eigenen Classification

Um eine eigene *Classification* als Grundlage für die Generierung der Defaults zu benutzen, muss eine Datenbank-Konfigurationsdatei, welche auf ein Schema verweist, das eine `classification`-Tabelle mit der gewünschten *Classification* besitzt, angelegt werden. Diese Datei würde bei Benutzung der Test-Datenbank des *OHDM*-Projekts wie in Listing 9.5 aussehen. Es wird angenommen, dass der Name dieser Datei „datasource-config.json“ ist und sich im selben Verzeichnis wie das Artefakt der Software befindet. Der Aufruf des Programms sieht dann wie folgt aus:

```
$ java -jar ohdm-style-generator-1.0.jar -d -db datasource-config.json -o styles my-styles.ohdmconfig
```

Die generierten Styles befinden sich anschließend im „styles“-Verzeichnis.

Benutzung der Standard-Classification

Wenn keine eigene *Classification* verwendet werden soll, können durch Benutzung der im Projekt definierten Standard-*Classification* (siehe Abschnitt 5.2) trotzdem Defaults generiert werden. Dazu ist nur das Setzen des „generate-defaults“-Parameters nötig:

```
$ java -jar ohdm-style-generator-1.0.jar -d -o styles my-styles.ohdmconfig
```

Die generierten Styles befinden sich anschließend im „styles“-Verzeichnis.

9.1.4 Erstellung von Styles und Konfiguration mit Upload auf GeoServer

Um die automatisierte Konfiguration des *GeoServers* zu ermöglichen wird eine *GeoServer*-Konfigurationsdatei benötigt. Bei Benutzung des *OHDM-GeoServers* würde diese Datei ungefähr wie in Listing 9.6 aussehen, wobei die Werte für *workspaceName*, *namespaceName*, *storeName* und *crsCode* beliebig ausgetauscht werden können.

Weiterhin wird vorausgesetzt, dass der in der *GeoServer*-Konfigurationsdatei angegebene Benutzer zum Lesen und Schreiben von Dateien berechtigt ist. Um diese Berechtigung für den *OHDM-GeoServer* zu erlangen genügt es, den Benutzer zur „tomcat7“-Gruppe hinzuzufügen.

Hinweis Damit die Änderungen nach erfolgreichem Upload der Dateien übernommen werden, muss über das *Web Administration Interface* des *GeoServers* die Konfiguration neu geladen werden. Dazu muss nach erfolgreichem Einloggen in der Navigationsleiste am linken Bildschirmrand zu „Server Status“ navigiert und in der Übersicht bei „Configuration and catalog“ auf „Reload“ geklickt werden.

An dieser Stelle soll angemerkt werden, dass bei den folgenden Szenarien die Generierung von Defaults durch Setzen des „generate-defaults“-Parameters unterstützt wird.

Erstellung von Styles ohne Layer-Konfiguration

Es ist möglich die generierten Styles ohne eine Verknüpfung mit *Layern* zu konfigurieren. Dabei wird keine Datenbank-Konfiguration benötigt.

Darüber hinaus ist es möglich zu entscheiden, ob die Styles im globalen oder im Style-Verzeichnis eines *Workspaces* platziert werden sollen. Wenn die Styles einem *Workspace* zugeordnet werden sollen, müssen in der *GeoServer*-Konfigurationsdatei die Parameter *workspaceName* und *namespaceName* gesetzt werden. Andernfalls müssen beide Felder nicht definiert sein. Die Konfiguration wird dann wie folgt gestartet:

```
$ java -jar ohdm-style-generator-1.0.jar -c geoserver-config.json my-styles.ohdmconfig
```

Erstellung von Styles mit Layer-Konfiguration

Um eine vollständige Konfiguration durchzuführen werden sowohl *GeoServer*- als auch Datenbank-Konfigurationsdatei benötigt. In der *GeoServer*-Konfigurationsdatei müssen bis auf den *crsCode* alle Parameter gesetzt werden. Die Konfiguration wird dann wie folgt gestartet:

```
$ java -jar ohdm-style-generator-1.0.jar -c geoserver-config.json -db datasource-config.json my
-styles.ohdmconfig
```

9.2 Listings

```
[boundary] {
    [undefined] {
        [>10] {
            useStyle = SimplePolygon, SimplePoint, SimpleLine
        }
        [<10] {
            useStyle = PolygonWithStyledLabel, PointWithLabel, SpacedLineWithLabel
        }
    }
    [historic] {
        [default] {
            useStyle = SimplePolygon, SimplePoint, SimpleLine
        }
    }
}
[military] {
    sameAs = boundary
}
<SimplePolygon> {
    * {
        fill: #000080;
        fill-opacity: 0.5;
        stroke: #FFFFFF;
        stroke-width: 2;
    }
}
<PolygonWithStyledLabel> {
    * {
        fill: #40FF40;
        stroke: white;
        stroke-width: 2;
        label: [name];
        font-family: Arial;
        font-size: 11px;
        font-style: normal;
        font-weight: bold;
        font-fill: black;
        label-anchor: 0.5 0.5;
        label-auto-wrap: 60;
        label-max-displacement: 150;
    }
}
<SimplePoint> {
    * {
        mark: symbol(triangle);
        mark-size: 12;
        :mark {
            fill: #009900;
            fill-opacity: 0.2;
            stroke: black;
            stroke-width: 2px;
        }
    }
}
<PointWithLabel> {
    * {
        mark: symbol(circle);
        mark-size: 6px;
        label: [name];
        font-fill: black;
        font-family: Arial;
        font-size: 12;
        font-weight: bold;
        label-anchor: 0.5 0;
        label-offset: 0 5;
        :mark {
            fill: red;
        }
    }
}
<SimpleLine> {
    * {
        stroke: blue;
        stroke-width: 3px;
        stroke-dasharray: 5 2;
    }
}
<SpacedLineWithLabel> {
    * {
        stroke: symbol(circle);
        stroke-dasharray: 4 6;
        label: [name];
        label-follow-line: true;
        label-max-angle-delta: 90;
        label-max-displacement: 400;
        label-repeat: 150;
        :stroke {
            size: 4;
            fill: #666666;
            stroke: #333333;
            stroke-width: 1px;
        }
    }
}
```


Listing 9.1: Beispiel für eine simple OHDM-Konfigurationsdatei

```
@Command(description = "Generates styles for the OHDM GeoServer based on a configuration
ohdmConfigFile.",
    name = "ohdm-style-generator", mixinStandardHelpOptions = true, version = "ohdm-style-
generator 1.0")
public class GeneratorCommand implements Callable<Integer> {

...

@Parameters(arity = "1", index = "0", description = "The config file that contains the
configuration for the generation of the styles.")
File ohdmConfigFile = null;

@Option(names = {"-o", "--output-dir"}, description = "The relative or absolute path to the
directory where the output will be stored.")
File outputDirectory = DEFAULT_OUTPUT_DIRECTORY;

@Option(names = {"-db", "--db-config"}, description = "The relative or absolute path to a .
json file containing the parameters for the database connection.")
File databaseConfigFile = null;

@Option(names = {"-c", "--geoserver-config"}, description = "The relative or absolute path to
a .json file containing the parameters for the connection to the GeoServer server.")
File connectionConfigFile = null;

@Option(names = {"-s", "--split-geometries"}, description = "Defines whether geometries should
 be split into 'lines', 'points' and 'polygons'")
boolean splitGeometries = true;

...
```

Listing 9.2: Ausschnitt der „GeneratorCommand“-Klasse

```
{
  "host": "myGeoServerHost",
  "user": "myUser",
  "password": "myPassword",
  "path": "/usr/share/geoserver/data_dir",

  "workspaceName": "ohdm-style-generator-test",
  "namespaceName": "ohdm_style_t",
  "storeName": "ohdm-test-db",
  "crsCode": "EPSG:3857"
}
```

Listing 9.3: GeoServer-Konfiguration mit Beispielwerten

```
{
    "host": "myDBHost",
    "database": "myDBName",
    "port": 5432,
    "user": "myUser",
    "password": "myPassword",
    "schema": "public"
  }
```

Listing 9.4: Konfiguration der Datenquelle mit Beispielwerten

```
{
  "host": "ohm.f4.htw-berlin.de",
  "database": "ohdm_test",
  "port": 5432,
  "user": "REPLACEME",
  "password": "REPLACEME",
  "schema": "public"
}
```

Listing 9.5: Beispiel für Datenbank-Konfiguration im OHDM-Projekt

```
{
  "host": "ohm.f4.htw-berlin.de",
  "user": "REPLACEME",
  "password": "REPLACEME",

  "path": "/var/lib/tomcat7/webapps/geoserver/data/",
  "workspaceName": "ohdm-style-generator-test",
  "namespaceName": "ohdm_style_t",
  "storeName": "ohdm-test-db",
  "crsCode": "EPSG:3857"
}
```

Listing 9.6: Beispiel für GeoServer-Konfiguration im OHDM-Projekt

Literaturverzeichnis

[For04] B. Ford. „Parsing Expression Grammars: A Recognition-based Syntactic Foundation“. In: *SIGPLAN Not.* 39.1 (Jan. 2004), S. 111–122. ISSN: 0362-1340. DOI: 10.1145/982962.964011. URL: http://doi.acm.org/10.1145/982962.964011 (zitiert auf S. 19, 20).

[Leg13] Legion of the Bouncy Castle Inc. *bouncycastle.org*. [Online; Zugriff am 23.08.2019]. 2013. URL: http://www.bouncycastle.org/documentation.html (zitiert auf S. 51).

[Ope17] OpenStreetMap Wiki. *About OpenStreetMap — OpenStreetMap Wiki*. [Online; Zugriff am 19.08.2019]. 2017. URL: https://wiki.openstreetmap.org/w/index.php?title=About_OpenStreetMap&oldid=1449350 (zitiert auf S. 17).

[Ope19a] Open Geospatial Consortium Inc. *Styled Layer Descriptor | OGC*. [Online; Zugriff am 20.08.2019]. 2019. URL: http://www.opengeospatial.org/standards/sld (zitiert auf S. 18).

[Ope19b] Open Source Geospatial Foundation. *About GeoTools - GeoTools*. [Online; Zugriff am 15.08.2019]. 2019. URL: https://geotools.org/about.html (zitiert auf S. 42).

[Ope19c] Open Source Geospatial Foundation. *CSS Styling - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/styling/css/index.html (zitiert auf S. 19).

[Ope19d] Open Source Geospatial Foundation. *Feature types - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/rest/api/featuretypes.html (zitiert auf S. 19).

[Ope19e] Open Source Geospatial Foundation. *Layers - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/data/webadmin/layers.html (zitiert auf S. 19).

[Ope19f] Open Source Geospatial Foundation. *Overview - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/introduction/overview.html (zitiert auf S. 18).

[Ope19g] Open Source Geospatial Foundation. *Publishing a PostGIS Table*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/master/en/user/gettingstarted/postgis-quickstart/index.html (zitiert auf S. 19).

[Ope19h] Open Source Geospatial Foundation. *Stores - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/data/webadmin/stores.html (zitiert auf S. 19).

[Ope19i] Open Source Geospatial Foundation. *Styling mixed geometry types - GeoServer 2.15.x User Manual*. [Online; Zugriff am 14.08.2019]. 2019. URL: https://docs.geoserver.org/stable/en/user/styling/sld/tipstricks/mixed-geometries.html (zitiert auf S. 46).

[Ope19j] OpenStreetMap Wiki. *Map Features — OpenStreetMap Wiki*. [Online; Zugriff am 19.08.2019]. 2019. URL: https://wiki.openstreetmap.org/w/index.php?title=Map_Features&oldid=1879758 (zitiert auf S. 18).

[Ope19k] OpenStreetMap Wiki. *Zoom levels — OpenStreetMap Wiki*. [Online; Zugriff am 19.08.2019]. 2019. URL: https://wiki.openstreetmap.org/w/index.php?title=Zoom_levels&oldid=1888272 (zitiert auf S. 18).

[Sch15] T. Schwotzer. *About OHDM - OpenHistoricalDataMap/OHDM-Documentation Wiki*. [Online; Zugriff am 20.08.2019]. 2015. URL: https://github.com/OpenHistoricalDataMap/OHDM-Documentation/wiki/About-OHDM (zitiert auf S. 17).

[Sch17] T. Schwotzer. *Home - OpenHistoricalDataMap/OHDM-Documentation Wiki*. [Online; Zugriff am 20.08.2019]. 2017. URL: https://github.com/OpenHistoricalDataMap/OHDM-Documentation/wiki (zitiert auf S. 17).

[Som12] I. Sommerville. *Software Engineering, 9., aktualisierte Auflage*. Pearson Deutschland GmbH, 2012. ISBN: 9783868940992 (zitiert auf S. 22).

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

_____

Datum, Ort, Unterschrift