NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Praxisbericht Nr. 5

Prüfling (Name)	Niels Gundermann
Matrikel-Nr.	5023
Studiengang, Zenturie	Wirtschaftsinformatik, I11b
Thema	Entwicklung einer Grammatik für eine DSL mit xText am Beispiel einer Sprache zur Definition von Pflichtprüfungen in profil c/s
Ausbildungsbetrieb	data experts gmbh
Prüfer/in	Dr. Stephan Neuthe
E-Mail der/ des Prüfers/in	stephan.neuthe@data-experts.de
Datum der Themenausgabe	23.10.2014
Datum der Abgabe	20.11.2014

fristgerechte Abgabe[1]	Ja ☒	Nein ☐
Sperrvermerk[2]	Ja ☐	Nein ☒
bestanden[3]	Ja ☒	Nein ☐

Datum 4.12.2014 Unterschrift Prüfer/in

[1] Die Bearbeitungsdauer für Praxisberichte soll vier Wochen nicht überschreiten (vgl. Merkblatt Praxisberichte). Wenn der Praxisbericht nicht innerhalb der vereinbarten Frist abgegeben wurde, dann gilt die Studienleistung als nicht bestanden.
[2] Der Praxisbericht enthält interne, vertrauliche Daten, die nicht zur Weitergabe an Dritte bestimmt sind.
[3] Als Bewertung kommen nur „bestanden" oder „nicht bestanden" in Betracht. Prüfer nutzen hierfür das beigefügte Bewertungsschema.

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Eidesstattliche Erklärung des Studenten / der Studentin

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder von mir noch von jemand anderem als Prüfungsleistung vorgelegt. Der von mir angegebene Prüfer hat die Arbeit zur Kenntnis genommen und als „bestanden" bewertet.

Datum 04.12.2014 ..... Unterschrift Student/in .....

Köllner Chaussee 11, 25337 Elmshorn, Telefon 0 41 21/40 90-0, Telefax 0 41 21/40 90 40

Praxisbericht: Entwicklung einer Grammatik für eine DSL mit xText am Beispiel einer Sprache zur Definition von Pflichtprüfungen in profil c/s

Niels Gundermann

26. November 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

Bei profil c/s handelt es sich um eine Client-Server-Anwendung, welche von der deg entwickelt wird und die Fördermittelveraltung in der Landwirtschaft abbildet. Diese Software ermöglicht die Bearbeitung von Förderanträgen. Im Laufe dieser Bearbeitung eines Förderantrags müssen bestimmte Prüfungen durchgeführt werden, die bei Fehlschlag die Ausschüttung der Fördermittel für diesen Antrag beeinträchtigen oder sogar verhindern. Diese Prüfungen können zum Teil in den Ämtern von den Sachbearbeitern selbst konfiguriert werden. Darüber hinaus gibt es auch Prüfungen, die bei der Bearbeitung von bestimmten Anträgen durchgeführt werden müssen und nicht in den Ämtern konfiguriert werden. Diese Prüfungen werden als *Pflichtprüfungen* bezeichnet und werden von der deg konfiguriert und in das Programm eingepflegt. Zur Konfiguration von Pflichtprüfungen wird in der deg eine Domain Specific Language (DSL) verwendet. In diesem Praxisbericht wird erläutert, warum DSLs eingesetzt werden und wie eine Grammatik grundsätzlich aufgebaut ist. Ferner wird die von der deg entwickelte DSL zur Konfiguration von Pflichtprüfungen untersucht und die Eigenschaften dieser, sowie die zugrundeliegende Grammatik erklärt. Zum Schluss wird auf die Umsetzung dieser Grammatik mit xText eingegangen.

2. Grundlegendes

2.1. Was ist eine DSL?

Unabhängig davon, ob man über DSL oder allgemeine Programmiersprachen (General Purpose Programming Language - GPL) spricht, haben Sprachen folgende vier Bestandteile:

1. Konkrete Syntax

   Die konkrete Syntax beschreibt die Notation der Sprache. Demnach bestimmt sie, welche Sprachkonstrukte der Nutzer einsetzen kann, um ein Programm in dieser Sprache zu schreiben. Die konkrete Syntax wird in so genannten Parse-Bäumen (konkrete Syntaxbäume) dargestellt. [Aho08, Seite 87]

2. Abstrakte Syntax

   Die abstrakte Syntax ist eine Datenstruktur, welche die Kerninformationen eines Programms beschreibt. Sie enthält keinerlei Informationen über Details bezüglich der Notation. Zur Darstellungs dieser Datenstruktur werden abstrakte Syntaxbäume genutzt.(Beispiel siehe [MSL+13, Seite 179]).

3. Statische Semantik

   Die statische Semantik beschreibt die Menge an Regeln bezüglich des Typ-Systems, die ein Programm befolgen muss. [MSL+13, Seite 26]

4. Ausführbare Semantik

   Die ausführbare Semantik ist abhängig vom Compiler. Sie beschreibt wie ein Programm zu seiner Ausführung funktioniert. [MSL+13, Seite 26]

Eine DSL ist eine Sprache, die für ein bestimmtes Problem - Domain/Domäne - optimiert ist. Man spricht auch von Meta-Sprachen oder Meta-Programmierung. Die Abstraktion dieser Sprache ist ebenso wie die Syntax an die Domäne angepasst. [MSL+13, Seite 28]
Mit einer GPL hingegen ist es möglich alle Probleme zu lösen, die mit einer Turing Maschine lösbar sind. So gesehen sind alle GPLs untereinander austauschbar. [MSL+13, Seite 27]
DSL werden in internen und externen DSLs unterteilt. Interne DSLs sind in einer GPL integriert. In der Regel orientieren sich die internen DSLs an dem

Typ-System der entsprechenden GPL, in der sie integriert sind. [MSL+13, Seite 50]

Externe DSLs sind von GPLs entkoppelt. Sie haben ein eigenes Typ-System und können nur mit Generatoren oder Interpreter[1], die auf sie abgestimmt sind, zur Ausführung kommen.

2.2. Warum werden DSLs eingesetzt?

Da DSLs ebenso wie GPLs von Rechnern verstanden werden und in einem IT-Unternehmen für mindestens eine GPL Know-How vorhanden sein sollte, stellt sich die Frage, warum ein neues Problem überhaupt mit einer DSL gelöst werden sollte. Letztendlich werden Ausdrücke einer DSL mittels Generatoren (Compiler) oder Interpretern auf Ausdrücke herunter gebrochen, die auf bestimmten Zielplattformen ausführbar sind. Eine GPL muss ebenfalls interpretiert oder übersetzt (compiled) werden.

Eine Gegenüberstellung von GPL und DSL kann aus Tabelle 1 entnommen werden.

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Tabelle 1: Domain-specific languages versus programming languages [MSL+13, Seite 31]

Dieser Tabelle ist zu entnehmen, dass ein Programm, welches mit einer DSL

[1]Ein Interpreter ist ein laufendes Programm auf der Zielplattform, welches das DSL Programm auf dieser lädt, damit es ausgeführt werden kann. Ein Generator hingegen wandelt das DSL Programm in Artefakte - oft GPL Code - um, womit die Ausführung direkt auf der Zielplattform stattfinden kann. [MSL+13, Seite 27]

geschrieben wurde, aufgrund der Größe der Sprache und der spezialisierung auf eine Domäne leichter zu warten ist, als ein Programm, welches mit einer GPL geschrieben wurde, und die gleichen Funktionen bereitstellt. Voraussetzung dafür ist, dass die DSL auch wirklich der entsprechenden Domäne angepasst ist. Weitere Vorteile der Nutzung von DSLs, die auch zum Teil aus Tabelle 1 abgeleitet wurden, sind die folgenden:

- **Weniger Lines of code**
- **Bessere Qualität, aufgrund kleinerer, unnötiger Freiheitsgrade**
- **Unabhängigkeit von der Technologien**
- **Kommunikationswerkzeug aufgrund der Eindeutigkeit der Sprachkonstrukte**
- **Unterstützende Tools sind vorhanden**

[MSL+13, Seite 40 ff.]

Bevor es zum Einsatz einer DSL kommen kann, muss diese jedoch entwickelt werden. Das bringt einige Nachteile mit sich:

- **Hoher Aufwand zur Erzeugung einer DSL**
- **Hoher Erfahrungsschatz benötigt**
- **Nutzung der DSL, obwohl sie noch nicht fertiggestellt wurde**
- **Neue DSL werden zu schnell entworfen**

[MSL+13, Seite 44]

Durch eine hohe Disziplin bei den Entwicklern und einer gut geplanten Entwicklungsphasen für die DSL, lassen sich meiner Meinung nach Nachteile wie *Nutzung der DSL, obwohl sie noch nicht fertiggestellt wurde* und *Neue DSL*

werden zu schnell entworfen vernachlässigen.

Ein wichtiger Punkt ist jedoch die Nutzung einer DSL als Kommunikationswerkzeug. Die Kunden können somit stärker in den Analyse- und Entwicklungsprozess mit einbezogen werden, weil sie bestimmte Sprachkonstrukte bzw. Bezeichnungen aus ihrem Arbeitsumfeld in die Syntax der DSL übernommen wurden. Das bietet eine Kommunikationsbasis, die eindeutiger ist, als die deutsche Sprache. In bestimmten Fällen können dadruch sicher Teile der Programmierung von Kunden übernommen werden. Hier fallen für das Unternehmen dann nur noch Schulungen und Konsultationen an. Inwiefern das ein wirtschaftlicher Nutzen oder Schaden wäre, muss induviduell geprüft werden. Auf jeden Fall würde es meiner Meinung nach durch die verbesserte Kommunikation zu einer Arbeitserleichterung kommen.

2.3. Grammatiken

Mit Hilfe von Grammatiken kann die Syntax einer Sprache beschrieben werden. [Hed12, Seite 26] Die Definition einer Grammatik ist wie folgt:

> Eine Grammatik ist ein Quadrupel $G = (V_N, V_T, P, S)$ mit
>
> 1. V_N, V_T sind endliche, nichtleere Mengen mit $V_N \cap V_T = \varnothing$
>
> V_N ist die Menge der Variablen (nonterminalen Symbole).
>
> V_T ist die Menge der terminalen Symbole (Terminale).
>
> 2. P ist eine endlichen Menge von Regeln der Form $a \rightarrow b$ mit $a \in (V_N \cup V_T)^+$, $b \in (V_N \cup V_T)^*$[2] Die Elemente von P werden *Produktionsregeln* oder *Grammatikregeln* genannt.
>
> 3. $S \in V_N$ ist das *Startsymbol*

[2] V^* bezeichnet die Menge alle Wörter über V. V^+ bezeichnet V^* ohne das leere Wort (ϵ). [Hed12, Seite 6]

Weiterhin gibt es verschiedene Formen von Grammatiken, die hierarchischen unterteilt werden. Diese Hierarchie nennt sich *Chomsky-Hierarchie* und wird wie folgt definiert:

> Sei $G = (V_N, V_T, P, S)$ eine Grammatik, $V = V_N \cup V_T$
>
> G ist vom
>
> **Typ 0:**
>
> falls keine Einschränkung vorliegt.
>
> **Typ 1:** oder kontextsensitiv
>
> falls jede Produktion die Form $a_1 A a_2 \rightarrow a_1 B a_2$ hat, wobei $A \in V_N, a_1, a_2 \in V^*, B \in V^+$, mit einer Ausnahme: $S \rightarrow \epsilon$
>
> **Typ 2:** oder kontextfrei
>
> falls jede Produktion die Form $A \rightarrow B$ hat, wobei $A \in V_N, B \in V^+$ mit derselben Ausnahmeregelung wie in Typ 1.
>
> **Typ 3:** oder regulär
>
> falls jede Produktion die Form $A \rightarrow aB$ oder $A \rightarrow a$ hat, wobei $A, B \in V_N, a \in V_T$, mit derselben Ausnahmeregelung wie in Typ 1.

Eine weitere Eigenschaft von Produktionsregeln ist die Ableitungsrichtung. Diese Eigenschaft spielt beim Aufbau des abstraken Syntaxbaums eine wichtige Rolle.

> Eine Linksableitung (Rechtsableitung) ist eine Bildungsvorschrift, bei der in jeder Produktionsregel $v \in (V_N \cup V_T)^*$ immer die linkeste (rechteste) Variable ersetzt wird. [Gro03, Seite 52]

Sprachen können somit von rechts nach links (Rechtsableitung), oder von links nach rechts (Linksableitung) aufgebaut werden.
Bei Programmiersprachen handelt es sich um entscheidbare Sprachen. Das bedeutet, dass für jedes zugrunde liegende Wort entschieden werden kann, ob es

zu dieser Sprache gehört oder nicht. [nb07, Seite 81]. Entscheidbare Sprachen werden durch Grammatiken beschrieben, die mindestens vom Typ 2 (kontextfrei) sind. [Hed12, Seite 38] Da eine Sprache nur von einem Rechner interpretiert werden kann, wenn sie entscheidbar ist, wird im weiteren Verlauf auf kontextfreie Grammatiken Bezug genommen.

3. Analyse der eingesetzten DSL

3.1. Erläuterung des Problems

Bei dem gewählten Beispiel geht es um die Konfiguration von *Pflichtprüfungen*, deren Stellenwert in der Einleitung bereits beschrieben wurde. Die Konfiguration einer *Pflichtprüfungen* wird über eine Meta-Beschreibung der Prüfungskonfiguration in einer Konfigurationsdatei vorgenommen. Der Begriff Prüfung wird im weiteren Verlauf als Synonym für den Begriff *Pflichtprüfungen* verwendet. Die zur Erläuterung notwendigen Eigenschaften einer Prüfungskonfiguration sind Tabelle 2 zu entnehmen[3].

Eigenschaft	Typ
ID	**Long**
Aktionen	**List<Aktion>**
Klassenname	**String**
Kurzbezeichnung	**String**
Langtext	**String**
Sichtbarkeit	**boolean**
Wirkungen	**List<Wirkung>**

Tabelle 2: Eigenschaften einer Prüfungskonfiguration

Die Aktionen und Wirkungen stehen dabei in einer 1:1-Beziehung. Eine Bei-

[3]Hierbei handelt es sich um eine vereinfachte Darstellung der Eigenschaften. Sie ist nicht mit dem Aufbau entsprechender Objekte in profil c/s gleichzusetzen.

spiel für die Definition solcher Prüfungen zeigt der folgende Ausschnitt einer Konfigurationsdatei.

Listing 1: Beispielkonfiguration einer Prüfung

```
1 CodesAlle = 20000
2 72002 = 20000
3 DvAntragsArt.ERWANTRAG = 200000
4 PruefungAktion.20000.1 = AntragAblehnen
5 PruefungAktion.20000.2 = AntragBewilligen
6 PruefungKlassenname.20000 = DE.data_experts.profi.profilcs.antrag.aum.allg.business.pruefungen
    .PAAumFNNichtUnvollstaendigUndHatKeinAnederungsBlatt
7 PruefungKurzbezeichnung.20000 = AUM-Fl\u00E4chenmappe nicht unvollst\u00E4ndig und ohne \
    u00C4nderungsblatt
8 PruefungLangtext.20000 = AUM-Fl\u00E4chenmappe ist nicht unvollst\u00E4ndig und hat kein \
    u00C4nderungsblatt
9 PruefungWirkung.20000.1 = VERHINDERT_AKTION
10 PruefungWirkung.20000.2 = VERHINDERT_AKTION
```

Eine Prüfung kann speziellen Förderanträgen zugewiesen werden. Diese Zuweisung findet über die Prüfungs-ID und die ID des Antrags (Antrags-ID) statt (Zeile 2 im Beispiel). Förderanträge werden auch zu Antragsarten (DvAntragsArt) zusammengefasst. Eine Prüfung kann über die Prüfungs-ID und das Schlüsselwort der Antragsart allen Förderprogrammen zugewiesen werden, die der Antragsart zugeordnet sind. Folgende Antragsarten stehen zur Verfügung (Schlüsselwort in Klammern):

- Auszahlungsantrag (AUSZANTRAG)
- Erweiterungsantrag (ERWANTRAG)
- Neuantrag (NEUANTRAG)
- Verlängerungsantrag (VERLANTRAG)

3.2. Aufbau der Sprache

Der Aufbau dieser Sprache kann über eine Grammatik $G = (V_N, V_T, P, S)$ und bestimmte Annahmen beschrieben werden. Im Folgenden werden die Produktionsregeln der Grammatik beschrieben. Zur Verdeutlichung steht an geeigneter

Stelle, der entsprechende Ausdruck aus der Konfigurationsdatei, der mit den im Voraus genannten Produktionsregeln erzeugt werden kann.
Zur Vereinfachung wird die Menge der Terminale mit $V_T = (ASCII)^*$[4] beschrieben.
$V_N = (IDs \mid ID \mid ANTRAGSZUWEISUNG \mid ANTRAGSART$
$\mid ANTRAGIDs \mid ANTRAGSART \mid EIGENSCHAFT$
$\mid ATTRIBUT \mid ATTRIBUTNUMMER \mid ATTRIBUTWERT).$
In jeder Konfigurationsdatei müssen zuerst alle Prüfungs-IDs deklariert werden, die den Prüfungen zugeordnet werden, deren Konfiguration in dieser Datei beschrieben wird. Grund dafür ist, dass die Eigenschaften einer Prüfungskonfiguration (Siehe Tabelle 2) mithilfe der Prüfungs-ID definiert werden. Dies erfolgt über Verbindungen von Prüfungs-ID und einem entsprechenden Schlüsselwörtern.
Für die Deklaration der Prüfungs-IDs werden folgende Produktionsregeln verwendet.
$S \rightarrow CodesAlle = IDs$
$IDs \rightarrow ID, IDs$
$ID \rightarrow nummerischer\ Text$
Für folgendes Beispiel wurden 3 Prüfungs-IDs deklariert.

Listing 2: Beispiel für ID-Deklaration in einer Konfigurationsdatei

```
1 CodesAlle = 4,20001,20002
```

Den oben genannten Antragsarten können diese Prüfungs-IDs zugeordnet werden. Dabei wird die Annahme getroffen, dass für alle folgenden Ableitungen des nonterminalen Symbols *ID* nur die terminalen Symbole verwendet werden, die bei *CodesAlle* deklariert wurden. Die Zuweisung zu den Antragsarten geschieht dann über die folgenden Produktionsregeln, wobei die erste nur für den

[4] Alle Kombinationen von ASCII-Zeichen.

Übergang von der Deklaration der Prüfungs-IDs zu der Zuweisung zu den Antragsarten benötigt wird.

$IDs \rightarrow ID\ ANTRAGSZUWEISUNG$

$ANTRAGSZUWEISUNG \rightarrow DvAntragsArt.ANTRAGSART = ANTRAGIDs$

$ANTRAGSART \rightarrow AUSZANTRAG \mid ERWANTRAG$

$\mid NEUANTRAG \mid VERLANTRAG$

$ANTRAGIDs \rightarrow ID\ ANTRAGSZUWEISUNG$

$ANTRAGIDs \rightarrow ID,\ ANTRAGIDs$

Durch diese Produktionsregeln werden folgende Sprachkonstrukte erzeugt.

Listing 3: Beispiel von Zuweisungen von Prüfungen zu Antragsarten

```
1 DvAntragsArt.AUSZANTRAG = 20001
2 DvAntragsArt.ERWANTRAG  = 20002,4,20001
3 DvAntragsArt.NEUANTRAG  = 4,20001
4 DvAntragsArt.VERLANTRAG = 20002
```

Die jeweiligen Eigenschaften einer Prüfungeskonfiguration werden über entsprechende Schlüsselwörter, die Prüfungs-ID und eine laufende Nummer definiert. Bei den Eigenschaften *Prüfungsalgorithmus*, *Sichtbarkeit*, *Langtext* und *Kurztext* wird die laufende Nummer nicht benötigt.

Zu der laufenden Nummer wird eine weitere Annahme getroffen. Die laufende Nummer wird immer um eins erhöht und beginnt bei eins. Dafür gelten folgende Produktionsregeln (Die erste wird wieder für den Übergang benötigt).

$ANTRAGIDs \rightarrow ID\ EIGENSCHAFT$

$EIGENSCHAFT \rightarrow ATTRIBUT.ID.ATTRIBUTNUMMER =$

$ATTRIBUTWERT$

$EIGENSCHAFT \rightarrow ATTRIBUT.ID = ATTRIBUTWERT$

$ATTRIBUTWERT \rightarrow Text$

$ATTRIBUTWERT \rightarrow Text\ EIGENSCHAFT$

$ATTRIBUTNUMMER \rightarrow nummerischer\ Text$

Wobei $Text \in V_T \wedge nummerischer\ Text \in (0-9)^*$ Das Beispiel ähnelt dem zu

Beginn des Kapitels vorgestellten Ausschnitt aus einer Konfigurationsdatei.

Listing 4: Beispiel von Definitionen der Eigenschaften einer Prüfungskonfiguration

```
1 PruefungAktion.20000.1 = AntragAblehnen
2 PruefungAktion.20000.2 = AntragBewilligen
3 PruefungKlassenname.20000 = DE.data_experts.profi.profilcs.antrag.aum.allg.business.pruefungen
    .PAAumFNNichtUnvollstaendigUndHatKeinAnederungsBlatt
4 PruefungKurzbezeichnung.20000 = AUM-Fl\u00E4chenmappe nicht unvollst\u00E4ndig und ohne \
    u00C4nderungsblatt
5 PruefungLangtext.20000 = AUM-Fl\u00E4chenmappe ist nicht unvollst\u00E4ndig und hat kein \
    u00C4nderungsblatt
6 PruefungWirkung.20000.1 = VERHINDERT_AKTION
7 PruefungWirkung.20000.2 = VERHINDERT_AKTION
```

3.3. Eigenschaften der eigesetzten DSL

Bei dieser DSL handelt es sich um eine externe DSL. Sie ist von der Zielplattform (Java) völlig entkoppelt.
Die exakten Regeln für diese Syntax sind nicht dokumentiert. Lediglich ein paar Beispiele zur Beschreibung von Prüfungskonfigurationen existieren bei der deg. Durch den Einsatz einer DSL ist die Domäne für den Entwickler jedoch leichter zu verstehen. Das Beispiel zur Beschreibung einer Prüfungskonfigurationen reicht aus, um dem Entwickler in die Lage zu versetzen, eine solche Prüfungskonfigurationen zu definieren. Die Beschreibung einer Prüfungskonfiguration lehnt sich an den Aufbau der Java-Klasse für die zu erstellenden Prüfungskonfigurationen an.
Die Semantik (statische und ausführbare) wird durch den Generator beschrieben.
Das Typ-System besteht dabei aus drei Typen.

- Text
- nummerischer Text
- Flag

Bei *Text* handelt es sich um eine einfache Zeichenkette. *nummerischer Text* hingegen ist eine spezielle Form von *Text*, die nur nummerische Zeichen enthalten darf. Der dritte Typ *Flag* kann zwei Werte annehmen indem es entweder in der Konfigurationsdatei enthalten ist oder nicht. Ein Beispiel dafür liefert die Sichtbarkeit von Prüfungen in folgendem Ausschnitt.

Listing 5: Beispiel einer Konfiguration der Sichtbarkeit einer Prüfung

```
1 CodesAlle = 20000, 20001
2 PruefungSichtbar.20001 =
```

Die Prüfung mit der ID 20000 ist sichtbar (da das Attribut *PruefungSichtbar* nicht aufgeführt ist) und die Prüfung mit der ID 20001 ist unsichtbar.
Der Generator geht von der korrekten Umsetzung der Grammatik aus. Daher ist es für die Effizienz bei der Umsetzung solcher Prüfungskonfigurationen wichtig, dass die Sprachkonstrukte in der Konfigurationsdatei der Syntax entsprechen, mit der der Generator umgehen kann.

4. Die Arbeit mit xText

In diesem Kapitel wird nach einer kurzen Einführung zu xText, die Grammatik aus Kapitel 3.2 mit xText umgesetzt. Das hat den Vorteil, dass die Sprachkonstrukte in den Konfigurationsdateien validiert werden können, bevor der Generator überhaupt zum Einsatz kommt.

4.1. Grundlegendes

Bei xText handelt es sich um eine Open-Source-Lösung für einen *ANTLR*-basierten Parser- und Editorgenerator[5]. xText bietet große Unterstützung, wenn es um das Entwickeln von Grammatiken geht. Darüber hinaus ist xText auch in

[5]ANTLR steht für *Another Tool for Language Recognition*. Es handelt sich dabei um einen Parser-Generator, welcher zum Lesen, Verarbeiten, Ausführen oder Übersetzen von strukturierten Texten oder binären Daten genutzt werden kann. Weitere Informationen unter [Par14]

der Lage einen Editor für die erstellten Grammatiken zu erzeugen. [ML09, Seite 1] Weiterhin gibt es für xText ein Eclipse-Plugin, auf welches im folgenden Verlauf Bezug genommen wird.

4.2. Umsetzung der Grammatik

Die Grammatiken werden in der Erweiterten Backus-Naur-From (EBNF) beschrieben. Mithilfe der EBNF können kontextfreie Grammatiken sehr kompakt dargestellt werden. Die Grundlegende Schreibweise ähnelt der, die im Kapitel 2.3 zur Überführung von nonterminalen Symbolen (a) in nonterminale oder terminale Symbole (b) genutzt wurde ($a \rightarrow b$).[6] In xText sieht eine Produktionsregel wie folgt aus:

Listing 6: Beispiel für die Zuweisung von Symbolen in der EBNF

```
1 a: b;
```

Terminale Symbole werden im Folgenden bei xText durch Hochkommas gekennzeichnet ('b').

Mit einer solchen Grammatik wird bei xText nicht nur die konkrete Syntax festgelegt, sondern auch die abstrakte.[7] Ein einfaches Beispiel zeigt, wie sich die Menge der IDs der benötigten Prüfungen in einer solchen Grammatik umsetzen lässt.

Listing 7: Auszug aus der Grammatik für die Sprache zur Konfiguration von Pflichtprüfungen

```
1          ids+=UsedIDs
2 UsedIDs:
3          'CodesAlle=' ID_List;
4 ID_List:
5          (PRUEFUNG)*;
6 PRUEFUNG:
7          INT | ',' INT;
```

[6] Weiterführende Informationen zu EBNF [Hed12, Seite 38 ff.]

[7] genaueres ist [ML09, Seite 3] zu entnehmen

Bei *ids* handelt es sich um ein Attribut, welches im xText-Model[8] benutzt wird. Alle Attribute des Models werden nach dem Schlüsselwort *Model:* deklariert. Die IDs für die Prüfungen werden vorerst als *INT* definiert. Die Zwischenstufe *ID_List* wurde eingeführt, um die Trennung der einzelnen IDs durch ein Komma umzusetzen.

Durch weitere Ergänzungen von Regeln, erhält man eine Grammatik mit deren Hilfe es möglich ist, die Syntax einer Konfigurationsdatei (Stand Juli 2014) zu validieren. Diese Grammatik ist im Anhang B zu finden.

Dabei sind drei Probleme aufgetreten:

1. Die Regeln dürfen keine Linksrekursion beinhalten.[9] Grund dafür ist, dass xText einen LL(*)-Parser nutzt, welcher die Rekursion einer Linksableitung nur bis zu einer bestimmten Rekursionstiefe[10]. [114, Seite 68 ff.]

2. Die Syntax wird zwar validiert. Die Semantik jedoch nicht. So ist es möglich in den Zuweisungen der Eigenschaften einer Prüfungskonfiguration bestimmte Prüfungs-IDs zu verwenden, die bei der Deklaration aller Prüfungs-IDs in der Konfigurationsdatei gar nicht vorkommen.

3. Werden mehrere Wirkungen oder Aktionen einer Prüfung zugeordnet, ist es möglich, dass die Nummerierung dieser nicht mit 1 beginnt oder fortlaufend ist. Das würde beim generieren zu Inkonsistenzen führen, da bei der Suche in der Konfigurationsdatei immer bei 1 gestartet wird:

[8] Als Model wird hier eine komplette Prüfungskonfiguration angenommen. Die Datei bildet also das Model ab. Diese Herangehensweise hat bei der ersten Umsetzung der Grammatik mit xText verwirrt, da das Model in profil letztendlich eine Prüfung beschreibt und nicht die Konfiguration mehrerer Prüfungen.

[9] Ein nonterminales Symbol, welches abgeleitet wird, darf auf der linken Seiten nicht durch sich selbst abgleitet werden

[10] Weiterführende Informationen zu LL(*)- oder LL(k)-Parsern sind unter [Gro03, Seite 54. f]

Listing 8: Auszug aus dem Parser der deg.

```
1 public List<PruefKonfVordef> createPruefKonfVordef( Long pruefungsCode ) {
2     ArrayList<PruefKonfVordef> result = new ArrayList<PruefKonfVordef>();
3     int i = 1;
4     do {
5     ...
6       i++;
7     } while ( getKonfig().hasParam( "PruefungAktion." + pruefungsCode + "." + i ) );
8     return result;
9   }
```

4.3. Generierung eines Editors

Ist die Grammatik einer DSL für xText in der EBNF definiert, lässt sich sehr einfach eine Editor erzeugen. Bevor man ihn erzeugt, sollte die Dateiendung auf die der xText-Editor achtet, angepasst werden. Diese ist in der Datei *GenerateKonfigDSL.mwe2* zu finden. Dort sollte eine Variable namens *fileExtensions* mit dem Schlüsselwort *var* deklariert sein. Der String, der dieser Variablen zugewiesen ist, bestimmt die Dateiendung, der zu validierenden Dateien. Um anschließend den Editor zu erzeugen muss man mit der rechten Maustaste die Datei *GenerateKonfigDSL.mwe2* klicken und in dem Kontextmenü unter dem Punkt *Run As -> MWE2 Workflow* auswählen (siehe Abbildung 1). Danach kann das gesamte Projekt als Eclipse-Applikation ausgeführt werden (siehe Abbildung 2). Es öffnet sich eine neue Eclipse-Umgebung in der der von xText generierte Editor integriert ist (siehe Abbildung 3). Wird eine Datei mit der vorab eingestellten Dateiendung erstellt (hier *mydsl1*), kann die Validierung und die Codevervollständigung des Editors genutzt werden (siehe Abbildung 4).

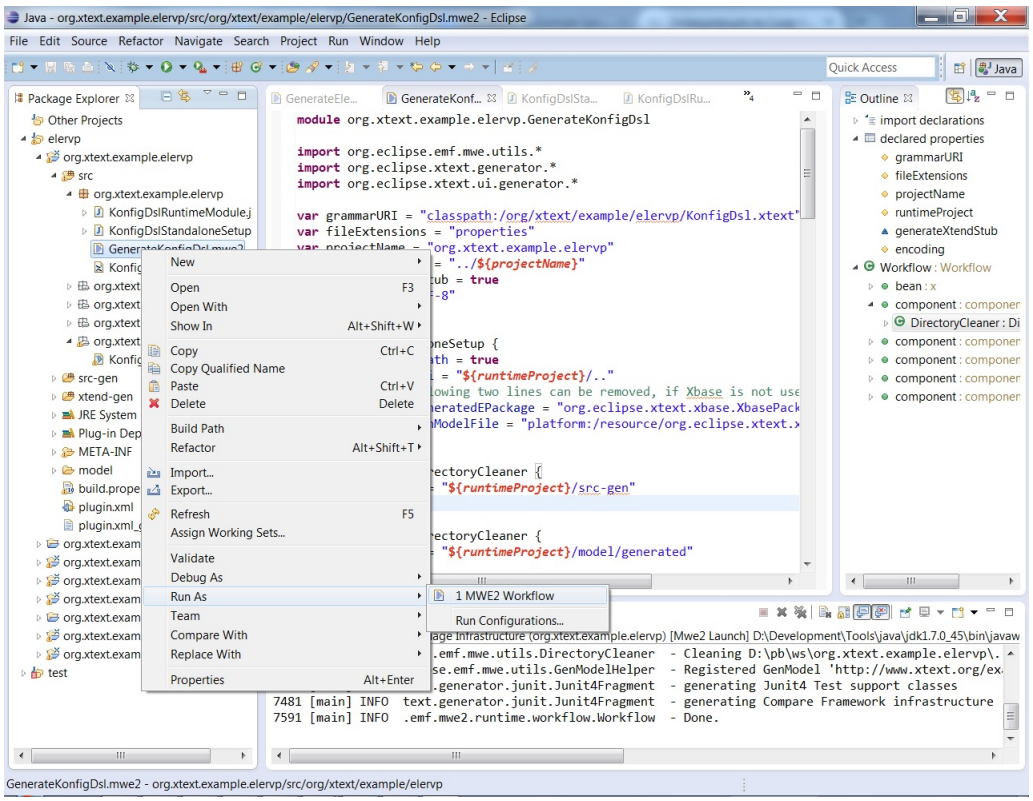

Abbildung 1: Screenshot 1

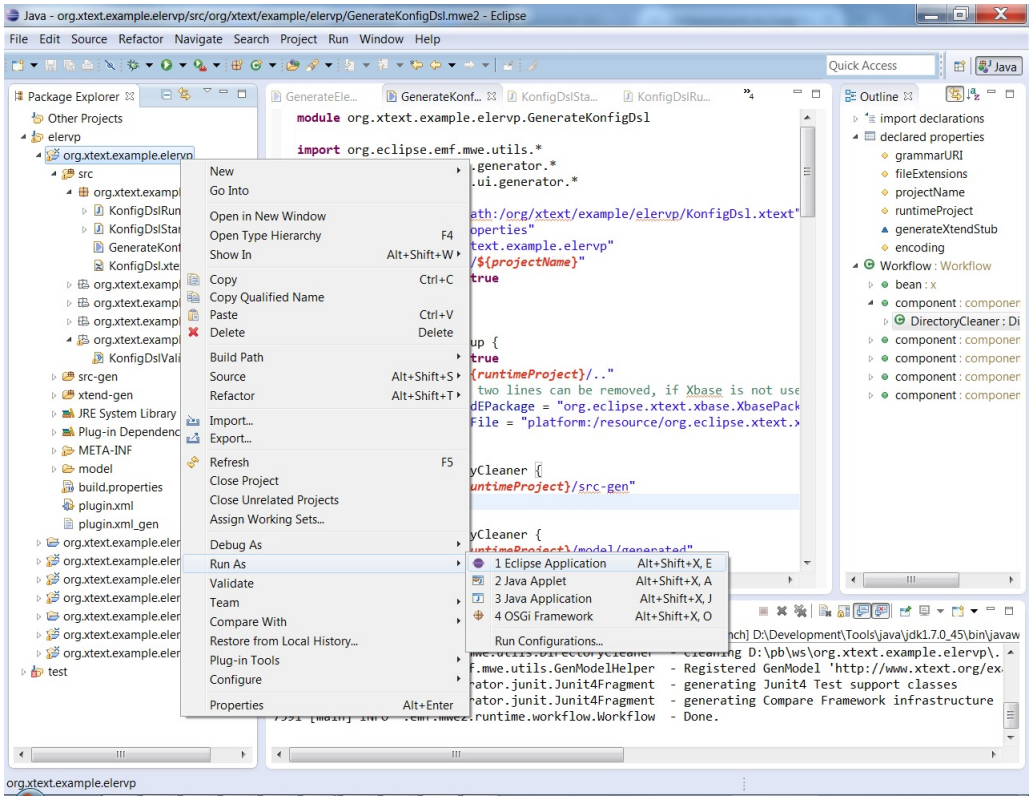

Abbildung 2: Screenshot 2

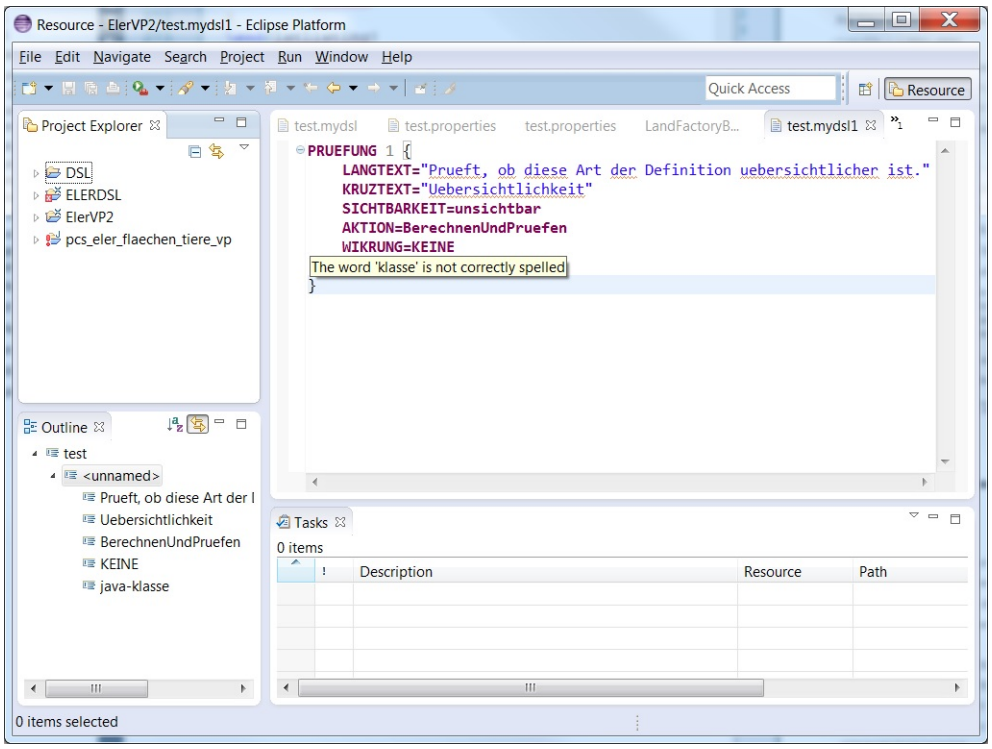

Abbildung 3: Screenshot 3

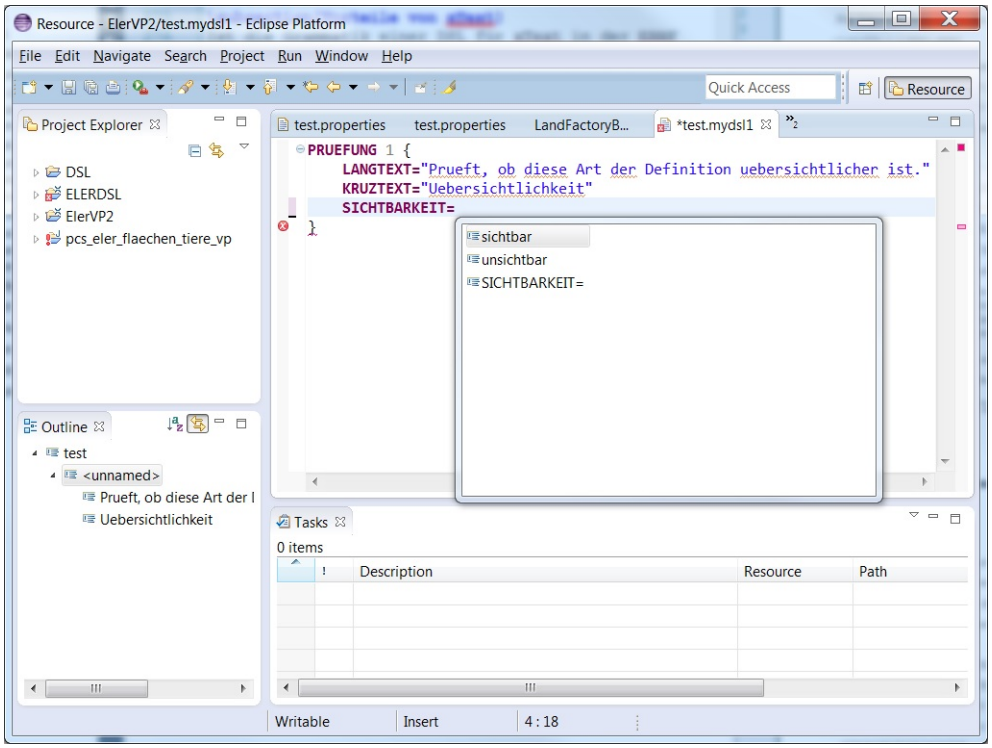

Abbildung 4: Screenshot 4

5. Verbesserungsvorschlag

Die Form der Beschreibung, die derzeit bei der deg genutzt wird, funktioniert solange die Konfigurationsdateien nur von den Mitarbeitern der deg gepflegt und überprüft werden. Diese Aufgabe könnte jedoch an die Kunden ausgelagert werden und somit die Mitarbeiter der deg entlasten. Allerdings müsste dafür eine andere Sprache entworfen werdenn, die eine Prüfungskonfiguration zusammenhängender beschreibt.

Grund dafür ist, dass meiner Meinung nach bei Prüfungskonfigurationen die Prüfung im Vordergrund steht. Demnach sollte diese innerhalb der Konfigurationsdatei in Gänze erfassbar sein.

Somit könnte neben der Erschaffung einer einheitlichen Struktur in der Konfigurationsdatei auch die Zuordnung der Eigenschaften über IDs eleminiert werden. Das hat den Vorteil, dass die Datei weitaus übersichtlicher wird. Bleibt die Zuordnung mittels IDs bestehen, tritt folgendes Problem auf.

Mit mehreren Prüfungen steigt auch die Anzahl der IDs und die der zu definierenden Eigenschaften der Prüfungskonfiguration. Ab einem gewissen Punkt sind diese Konfigurationsdateien meiner Meinung nach nicht mehr wartbar. Grund dafür ist vor allem die mangelnde Übersichtlichkeit.

In Anhang B befindet sich eine Grammatik, die eine Syntax definiert, mit der genau diese beiden Probleme gelöst werden. Sie ist so konzipiert, dass eine Definition einer Prüfungskonfiguration als xText-Model betrachtet wird. Die Konfigurationsdateien können weiterhin mehrere Prüfungen definieren. Die Form der Definition ändert sich jedoch. Mit der Syntax, die durch diese Grammatik definiert wird, kann eine Prüfungskonfiguration wie folgt dargestellt werden:

Listing 9: Konfiguration einer Prüfung durch eine Sprache der die neue Grammatik zugrunde liegt

```
1 PRUEFUNG 1 {
2         LANGTEXT="Irgendein␣Langtext."
```

```
3         KURZTEXT="Ein Kurztext"
4         SICHTBARKEIT=unsichtbar
5         AKTION=BerechnenUndPruefen
6         WIRKUNG=Keine
7         ALGORITHMUS="java-klasse"
8 }
```

Bei der Betrachtung der Grammatik im Anhang B fällt auf, dass diese weitaus übersichtlicher ist. Daher ist es einfacher die Grammatik zu erweitern. Weiterhin ist es auch leichter die Syntax über die Grammatik zu erlernen. Damit wäre die Grammatik mit einer Dokumentation gleichzusetzen.
Dieses resultierende Sprache weißt jedoch auch Schwächen auf. Beispielsweise kann jeweils eine Wirkung und Aktion für eine Prüfung definiert werden.

6. Fazit

Der Einsatz von Meta-Programmierung oder DSLs für dieses Problem zeigt einige Vorteile, die der Einsatz von DSLs mit sich bringt. Ich habe die Lesbarkeit/Übersichtlichkeit durchaus angefochten, dennoch ist sie besser im Vergleich zu einer Umsetzung mittels einer GPL.
Zu der Arbeit xText ist abschließend zu sagen, dass die Definition einer Grammatik mehrere Vorteile bietet.

1. Hilfe beim Schreiben des Meta-Code durch Validierung und Code-Vervollständigung.

2. Dokumentation der eingesetzten Sprache.

Die Grammatik, welche das bestehende Model beschreibt, kann durch spezielle Validierungen noch erweitert werden. Beispielsweise könnte man das zweite und dritte benannte Problem in Kapitel 4.2 durch zusätzliche Validierungsregeln umgehen.[11]

[11] Dafür sollte jedoch das Model so umgestellt werden, dass es die einzelnen Prüfungskonfigurationen umfasst und nicht die gesamte Konfigurationsdatei.

Die zweite Grammatik ist übersichtlicher. Jedoch ist der Generator, der bei der deg zum Einsatz kommt, nicht auf diese Art der Definition abgestimmt. Ein weiteres Problem dieser Grammatik ist die Ausgangslage. Die oben beschriebenen Eigenschaften sind vereinfacht dargestellt. Die Aktionen und deren Wirkungen werden nicht direkt in der Prüfung hinterlegt. Somit ist es technisch betrachtet nicht zweckmäßig diese Eigenschaften in der Meta-Beschreibung einer Prüfung mit einzubauen. Allerdings bietet die zusammenfassung von Eigenschaften einer Prüfungskonfiguration zu einem Block eine gewisse Übersichlichkeit.

Anhänge

A. Literaturverzeichnis

Literatur

[114] *Xtext Documentation*. URL: http://www.eclipse.org/Xtext/cumentation/2.6.0/Xtext Documentation.pdf, September 2014. Zuletzt eingesehen am 24.11.2014.

[Aho08] Aho, Alfred V: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.

[Gro03] Grosche, Günter: *Teubner-Taschenbuch der Mathematik*, Band 8. Vieweg+Teubner Verlag, 2003.

[Hed12] Hedtstück, Ulrich: *Einführung in die Theoretische Informatik*. Oldenbourg Wissenschaftsverlag, 2012.

[ML09] Markus Voelter und Lars Corneliussen: *Carpe Diem*. DotNetPro, 5 2009.

[MSL+13] Markus Voelter, Sebastian Benz, Lennart Kats, Mats Helander, Eelco Visser und Guido Wachsmuth: *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013.

[nb07] *Einführung in Formale Sprachen, Berechenbarkeit, Infomrations- und Lerntheorie*. Oldenbourg Wissenschaftsverlag GmbH, 2007.

[Par14] Parr, Terence: *antlr.org*. URL: http://www.antlr.org/, 2014.

B. Implementierung

Listing 10: Grammatik für die bestehende Sprache zur Konfiguration von Prüfungen

```
1 grammar org.xtext.example.elervp.KonfigDsl with
2 org.eclipse.xtext.common.Terminals
3
4 generate konfigDsl "http://www.xtext.org/example/elervp/KonfigDsl"
5
6 Model:
7         usedids+=UsedIDs &
8         (spezantragszuweisung+=SPEZ_ANTRAGSZUWEISUNG)* &
9         (antragszuweisung+=Zuweisung)* &
10        (pruefungsaktion+=PRUEFUNGSAKTION)* &
11        (pruefungsklassenname+=PRUEFUNGSKLASSENNAME)* &
12        (pruefungskurzbezeichnung+=PRUEFUNGSKURZBEZEICHNUNG)* &
13        (pruefungslangtext+=PRUEFUNGSLANGTEXT)* &
14        (pruefungswirkung+=PRUEFUNGSWIRKUNG)* ;
15
16 SPEZ_ANTRAGSZUWEISUNG:
17        INT 'ID_List;
18
19 PRUEFUNGSLANGTEXT:
20 'PruefungLangtext.'PRUEFUNG'='STRING;
21
22 PRUEFUNGSKURZBEZEICHNUNG:
23 'PruefungKurzbezeichnung.'PRUEFUNG'='STRING;
24
25 PRUEFUNGSKLASSENNAME:
26        'PruefungKlassenname.'PRUEFUNG'='STRING;
27
28 PRUEFUNGSAKTION:
29        'PruefungAktion.'AKTIONSID'='AKTION;
30
31 AKTION:
32        'BerechnenUndPruefen';
33
34 AKTIONSID:
35        PRUEFUNG'.'INT;
36
37 PRUEFUNGSWIRKUNG:
38        'PruefungWirkung.'WIRKUNGSID'='WIRKUNG;
39
40 WIRKUNG:
41        'VERHINDERT_AKTION';
42
43 WIRKUNGSID:
44        PRUEFUNG'.'INT;
45
46 Zuweisung:
47        'DvAntragsArt.'ANTRAGSART'='ID_List;
```

```
48
49 ANTRAGSART:
50         'AUSZANTRAG' | 'ERWANTRAG' | 'NEUANTRAG' | 'VERLANTRAG';
51
52 UsedIDs:
53         'CodesAlle=' ID_List;
54
55 ID_List:
56         (PRUEFUNG)*;
57
58 PRUEFUNG:
59         INT | ',' INT;
```

Listing 11: Vorgeschlagene Grammatik für eine Sprache zur Konfiguration von Prüfungen

```
1 grammar org.xtext.example.mydsl1.Elervp2 with
2 org.eclipse.xtext.common.Terminals
3
4 generate elervp2 "http://www.xtext.org/example/mydsl1/Elervp2"
5
6 Model:
7         (pruefung+=Preufung)*;
8
9 Preufung:
10         'PRUEFUNG ' name=INT '{'
11         'LANGTEXT=' langtext=Langtext
12         'KURZTEXT=' kurztext=Kurztext
13         'SICHTBARKEIT=' sichtbarkeit=Sichtbarkeit
14         ('AKTION=' aktionen=Aktionen)*
15         'WIKRUNG=' wirkung=Wirkung
16         'ALGORITHMUS=' algorithmus=Algorithmus
17         '}';
18
19 Sichtbarkeit:
20         sichtbar?='sichtbar' | unsichtbar?='unsichtbar';
21
22 Algorithmus:
23         klasse=STRING;
24
25 Wirkung:
26         bezeichnung='VERHINDERT_AKTION' | bezeichnung='KEINE';
27
28 Aktionen:
29         bup='BerechnenUndPruefen' | frei='Freigeben';
30
31 Kurztext:
32         name=STRING;
33
34 Langtext:
35         name=STRING;
```