Bachelorarbeit im Studiengang Angewandte Informatik an der Fakultät IV - Wirtschaft und Informatik Hochschule Hannover

Entwicklung eines generischen XML-Editors für ein interoperables Programmieraufgabenformat

Paul Reiser

6. April 2017

Autor Paul Reiser
apaulreiser@gmail.com

Erstprüfer: Prof. Dr. Robert Garmann
Abteilung Informatik, Fakultät IV
Hochschule Hannover
robert.garmann@hs-hannover.de

Zweitprüfer: Prof. Dr. Felix Heine
Abteilung Informatik, Fakultät IV
Hochschule Hannover
felix.heine@hs-hannover.de

Abstract

Diese Bachelorarbeit befasst sich mit der Entwicklung eines generischen XML-Editors für das ProFormA-Aufgabenformat. ProFormA ermöglicht einen Aufgabenaustausch zwischen Hochschulen, Lernmanagementsystemen und Gradern. Aufgaben werden von Lehrkräften genutzt und für ihren individuellen Lehrkontext angepasst. Weil das manuelle Editieren von ProFormA-Aufgaben durch XML, Erweiterbarkeit und Komplexität des Formats erschwert wird, muss ein XML-Editor entwickelt werden, der generische Mechanismen implementiert, die das Anzeigen, Editieren, Hinzufügen und Entfernen von ProFormA- und Fremdformatelementen ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

XML	Extensible Markup Language
XSD	XML Schema Definition
DTD	Document Type Definition
HTML	HyperText Markup Language
LMS	Learning Management System
URI	Uniform Resource Identifier
JAXB	Java Architecture for XML Binding
GUI	Graphical User Interface
URI	Uniform Resource Identifier

Glossar

ProFormA	Ein interoperables Austauschformat für Programmieraufgaben
Grader	Eine Software zur automatisierten Bewertung von Programmieraufgaben
Namensraum	Wird für eine eindeutige Identifizierung von XML-Elementen verwendet
Praktomat	Ein webbasierter Grader
Typ-Introspektion	Die Fähigkeit, Eigenschaften eines Datentyps während der Programmlaufzeit zu untersuchen

1. Einleitung

An Hochschulen werden sogenannte Grader eingesetzt, die zu vorgegebenen Programmieraufgaben Lösungen von Studenten entgegennehmen, bewerten und schließlich Feedback über die Korrektheit der Lösung zurückmelden. Programmieraufgaben werden von Aufgabenautoren erstellt und mit anderen Hochschulen ausgetauscht.

Für einen verbesserten Austausch von Programmieraufgaben wurde ein Aufgaben-Austauschformat namens ProFormA entwickelt. Dieses Format definiert, wie die Struktur einer Aufgabe in einem *Extensible Markup Language*-Dokument (XML) auszusehen hat. Aufgaben, die im ProFormA-Format vorliegen, sind erweiterbar und können um zusätzliche Informationen aus anderen Formaten ergänzt werden.

Lehrkräfte nutzen Aufgaben für Schulungszwecke und passen sie für ihren individuellen Lehrkontext an. Weil das manuelle Editieren von ProFormA-Aufgaben durch das XML-Format unnötig erschwert wird, muss ein Werkzeug erstellt werden, welches das Editieren von ProFormA-Aufgabe unterstützt.

1.1. Zielsetzung

Diese Bachelorarbeit befasst sich mit der Entwicklung eines XML-Editors, der das Editieren von ProFormA-Aufgaben erleichtert. Dabei muss geklärt werden, wie sich die Erweiterbarkeit von ProFormA-Aufgaben geeignet mit dem Editor realisieren lässt. Da Benutzerfreundlichkeit eine wichtige Voraussetzung für den Editor ist, müssen die Stärken und Schwächen von bestehenden Lösungen untersucht werden. Die Erkenntnisse sollen in die Entwicklung des XML-Editors miteinfließen.

1.2. Aufbau

In Kapitel 2 wird das ProFormA-Format vorgestellt. Dabei werden die einzelnen Elemente erläutert, die zusammengenommen eine ProFormA-Aufgabe ausmachen. In Kapitel 3 wird auf den aktuellen Zustand eingegangen. Es wird verdeutlicht, auf welche Weise das ProFormA-Format von Aufgabenautoren, Lehrkräften und Studenten genutzt wird. Darüber hinaus wird der ProFormA-Editor vorgestellt, der zur Zeit für die Bearbeitung von Aufgaben genutzt werden kann. Kapitel 4

beschäftigt sich mit der Analyse der bisherigen Situation und arbeitet Anforderungen aus, die an ein Werkzeug zum Editieren von ProFormA-Aufgaben gestellt werden. Kapitel 5 stellt das erarbeitete Konzept zur Lösung dieser Anforderungen vor. Die Implementierung wird anschließend in Kapitel 6 vorgestellt und in Kapitel 7 getestet.

2. Grundlagen

2.1. XML-Editor

Da das direkte Arbeiten mit XML sehr schwerfällig werden kann, gibt es XML-Editoren, die die Darstellung und Bearbeitung von XML-Dokumenten durch eine benutzerfreundliche Schnittstelle unterstützen. Editoren können sich dabei sogenannte XML Schemadefinitionen (XSD) zunutze machen und die Gültigkeit der Eingabedaten auf ihre Gültigkeit überprüfen. Ausmaß und Form der Unterstützung hängen dabei vom jeweiligen Editor ab. Die Funktionalität zum Editieren von XML beschränkt sich auf das, was in einem Editor zu sehen ist. In den meisten Fällen wird das zugrundeliegende XML-Format vor dem Benutzer verborgen. Dadurch kann man sich bei dem Editieren auf den wesentlichen Inhalt fokussieren, ohne vom Format abgelenkt zu werden.

2.2. ProFormA-Format

Im Rahmen des eCULT-Projekts ([7]) entstand ein erweiterbares Austauschformat namens *ProFormA* ([10]). Es handelt sich um eine XML-Schemadefinition, die beschreibt, wie Programmieraufgaben in XML aufgebaut sein müssen. Durch einen einheitlichen und festgeschriebenen Aufbau kann die Austauschbarkeit, Nutzbarkeit und automatisierte Weiterverarbeitung von Aufgaben verbessert werden.

Für eine Aufgabe im ProFormA-Format können sämtliche Informationen und Ressourcen, die für die Bereitstellung und Lösung einer Aufgabe notwendig sind, zu einer eigenständigen Einheit zusammengefasst werden ([11]).

2.2.1. Aufbau

Das ProFormA-Format beschreibt alle Informationen und Ressourcen, die Bestandteil einer Aufgabe (*task*) sind. Dazu zählen unter anderem Aufgabenbeschreibung und Musterlösungen. Eine Aufgabe beinhalte auch Informationen zu Prüfungen, denen studentische Lösungen zu der Aufgabe während einer automatisierten Bewertung durch einen Grader unterzogen werden ([10]). Durch die Erweiterbarkeit einer ProFormA-Aufgabe können weitere Informationen angefügt werden, die aus anderen XML-Schemadefinitionen stammen.

Es folgt eine Auflistung der Element, die eine ProFormA-Aufgabe ausmachen.

task (Aufgabe) Das Wurzelelement selbst hat drei Attribute. Eine Identifikationsnummer, eine Kennung der natürlichen Sprache, in der die Aufgabe verfasst ist, sowie eine optionale Referenz auf eine Elternaufgabe, aus der die Aufgabe entstammt.

description (Beschreibung) Die Aufgabenbeschreibung für eine Programmiersprache. Es steht Aufgabenautoren frei, Beschreibungen in einfachem Text oder in HTML bereitzustellen.

proglang (Programmiersprache) Enthält den Namen der Programmiersprache und die Version, in der die Aufgabe zu lösen ist.

submission-restrictions (Abgabeeinschränkungen) Die Abgabeeinschränkung steuert, in welcher Form studentische Lösungseinreichungen akzeptiert werden. Dateigrößen können zum Beispiel auf eine maximale Größe limitiert werden. Dateitypen können auf bestimmte Endungen beschränkt werden.

files (Dateien) Führt alle Dateien auf, die für die Bearbeitung einer Aufgabe relevant sind. Dies schließt sowohl die Bearbeitung einer Aufgabe von Studenten als auch die Verarbeitung einer Aufgabe von einem Grader ein. Der Inhalt einer Datei kann über einen Dateipfad verlinkt oder direkt in eine Aufgabe eingebettet werden.

external-resources (Externe Ressourcen) Externe Ressourcen referenzieren Ressourcen, die sich außerhalb des Aufgabenbündels befinden. Üblicherweise werden alle Dateien, die für eine Aufgabe relevant sind, zusammen mit der Aufgabe in einem eigenständigen ZIP-Archiv abgelegt. Bei manchen Ressourcen ist dies jedoch nicht immer möglich. Bei Aufgaben, deren Bearbeitung oder Installation von sehr großen Dateien abhängt, ist es nicht ratsam, die Dateien zusammen mit ins Archiv zu legen, da es den Aufgabenaustausch aufgrund der Archivgröße sonst unhandlich macht ([10]). In solchen Fällen werden Ressourcen als externe Ressourcen referenziert. Für Ressourcen, die einer komplexeren Referenzierungsmethode bedürfen, kann die Erweiterbarkeit einer Aufgabe genutzt werden. Dazu werden Elemente aus Namensräumen anderer Schemadefinitionen angebunden.

model-solutions (Musterlösungen) Enthält die Musterlösungen für eine Aufgabe. Musterlösungen sind Dateien, die korrekten Programmcode zur Lösung einer Aufgabe enthalten können.

tests (Prüfungen) Beschreibt die Prüfungen, die von Gradern durchgeführt werden, sobald eine Aufgabe automatisiert bewertet wird. Es gibt verschiedene Prüfungsarten. Zwei Beispiele wären Syntaxprüfungen und Stilprüfungen von Programmquellcode.

Eine Prüfung enthält alle notwendigen Konfigurationen, die für eine automatisierte Durchführung dieser Prüfung erforderlich sind.

grading-hints (Punktevergabe) Dieses Element ist dafür vorgesehen, Informationen zur Punktevergabe einer Aufgabe zu speichern. Da nicht festgeschrieben wird, wie diese Information aussehen muss, können an dieser Stelle Elemente aus anderen Namensräumen angebunden werden, die diese Information enthalten.

meta-data (Metadaten) Metadaten dienen dem Zweck, andere Daten zu beschreiben, was den Umgang mit diesen Daten erleichtern kann. An dieser Stelle können Elemente aus anderen Namensräumen angefügt werden, die Metadaten zu einer Aufgabe enthalten.

3. Ist-Zustand

Das ProFormA-Format kommt bereits in unterschiedlichen Systemen zum Einsatz. Aufgaben können zwischen Gradern ausgetauscht werden, indem sie in die graderspezifischen Systeme importiert oder aus ihnen exportiert werden ([11]). Ein Austausch von Aufgaben kann auch zwischen unterschiedlichen Lernmanagementsystemen (LMS) stattfinden. Um eine Aufgabe in einem LMS zur Verfügung zu stellen, muss die Aufgabe in das jeweilige LMS hochgeladen werden. Von dort aus kann sie von Gradern verwendet werden, die an das LMS angebunden sind.

An der Erstellung, Anpassung und Verwendung von ProFormA-Aufgaben beteiligen sich die folgenden drei Rollen:

Aufgabenautor Aufgabenautoren setzen neue Aufgaben auf. Bei der Aufgabenverfassung machen sie sich speziellen Werkzeugen zunutze. Für graderspezifische Aufgabenkonfigurationen können zusätzliche Schemadefinitionen für Namensraumerweiterungen bereitgestellt werden, die in den erweiterbaren Elementen des ProFormA-Formats Verwendung finden.

Lehrkraft Lehrkräfte nutzen Aufgaben für Schulungszwecke. Hierzu passen sie bestehende Aufgaben für ihren individuellen Lehrkontext an. Die Anpassung einer Aufgabe findet durch das manuelle Editieren des XML-Dokumentes statt, in dem eine Aufgabe vorliegt.

Student Studenten verwendet eine Aufgabe, indem sie Lösungen zur Aufgabe einreichen. In den meisten Fällen bekommen sie nur die Aufgabenbeschreibung einer Aufgabe vorgelegt. Mit dem ProFormA-Format selbst kommen Studenten nicht in Berührung.

3.1. Editieren von ProFormA-Aufgaben

ProFormA-Aufgaben liegen als ZIP-Archive vor, in denen alle Dateien enthalten sind, die für eine Aufgabe relevant sind. Eine Lehrkraft, die eine Aufgabe für Schulungszwecke verwenden und anpassen möchte, muss das ZIP-Archiv entpacken, die Aufgabe in Form einer XML-Datei editieren und anschließend wieder zu einem ZIP-Archiv zusammenfügen. Lehrkräfte kennen sich nicht immer mit XML aus. Das Editieren von Aufgaben wird durch die zusätzliche Komplexität

des ProFormA-Formats erschwert. Für die Lehrkraft entsteht dadurch ein Bedarf an einem Werkzeug, welches das Editieren von Aufgaben erleichtert.

3.2. ProFormA-Editor

An der Ostfalia Hochschule ist ein Editor namens ProFormA-Editor entstanden. Es handelt sich um eine Webanwendung, die in JavaScript entwickelt ist und in einem Internet-Browser aufgerufen und bedient werden kann. Der ProFormA-Editor ist auf die Erstellung und Bearbeitung von ProFormA-Aufgaben spezialisiert.

3.2.1. Funktionalität

Es folgt eine Auflistung der vorhandenen Funktionalität des ProFormA-Editors für die Versionsnummer 2.0.1 ([4]). Dabei wird ein besonderes Augenmerk auf die Unterstützung für die Elemente des ProFormA-Format gelegt, die in Unterabschnitt 2.2.1 beschrieben sind. Dadurch soll ein Überblick verschafft werden, in welchem Ausmaß der Editor das ProFormA-Format unterstützt.

task (Aufgabe) Das ID-Attribut einer Aufgabe wird automatisch vergeben. Die Angabe des Aufgabentitels sowie der verwendeten natürlichen Sprache ist möglich. Die ID der Elternaufgabe kann nicht konfiguriert werden.

description (Beschreibung) Das Editieren der Aufgabenbeschreibung wird unterstützt. HTML-Tags können für die Formatierung verwendet werden. Das Ergebnis wird in einer Live-Vorschau dargestellt.

proglang (Programmiersprache) Der Editor unterstützt die Kombinationen *Java/1.6*, *Java/1.8*, *Python/2.x* und *setlX/2.40*.

Programmiersprache und Version sind nicht frei wählbar. Abhängig von der gewählten Kombination werden Buttons für unterschiedliche Prüfungsarten auf der Oberfläche ein- und ausgeblendet.

submission-restrictions (Abgabeeinschränkungen) Eine Angabe für die Abgabeeinschränkung von studentische Lösungen ist möglich. Beschränkt können jedoch nur die Dateigröße und die Dateiendung von Lösungsdateien. Das ProFormA-Format bietet hierzu jedoch noch viele andere Möglichkeiten, Abgabeeinschränkungen zu definieren.

files (Dateien) Dateien können hinzugefügt und bearbeitet werden.

external-resources (Externe Ressourcen) Die Angabe von externen Ressourcen wird nicht unterstützt.

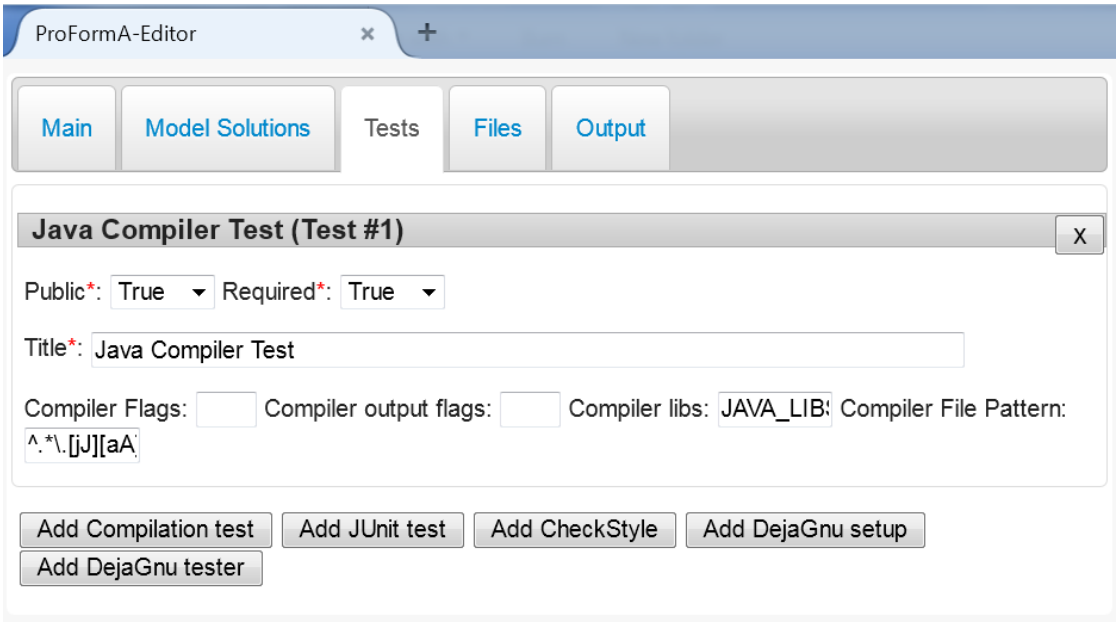

Abbildung 3.1.: ProFormA-Editor: Tests; Quelle: [4]

model-solutions (Musterlösungen) Musterlösungen können hinzugefügt und bearbeitet werden.

tests (Prüfungen) Das Editieren von Test-Elementen aus dem ProFormA-Namensraum wird nicht unterstützt. Stattdessen können vordefinierte Prüfungsarten für den Grader *Praktomat* als Erweiterung in die Aufgabe eingetragen werden (vgl. Abbildung 3.1).

Folgende Prüfungsarten werden angeboten:

- Java Compilation Test
- Java JUnit Test
- Java CheckStyle
- DejaGnu Setup
- DejaGnu Tester

Diese Prüfungsarten sind für den Grader Praktomat bestimmt. Dies macht sich durch das Anfügen des Praktomat-Namensraums in das XML-Dokument der Aufgabe bemerkbar. Eine Möglichkeit, Prüfungen für andere Grader an einen Task anzufügen, ist nicht gegeben.

grading-hints (Punktevergabe) Wird nicht unterstützt.

meta-data (Metadaten) Wird nicht unterstützt.

Existierende Aufgaben können vom Editor als XML-Dokumente geladen, bearbeitet und wieder als XML abgespeichert werden. Darüberhinaus können Aufgaben als ZIP-Archiv über den Internet-Browser auf die lokale Festplatte heruntergeladen werden. Dateien können per "drag and drop" in die Aufgabe eingefügt werden. Auf fehlende Pflichteingaben wird hingewiesen. Eine XML-Validierung wird für ProFormA-Schemadefinition angeboten, jedoch nicht für andere [4].

4. Analyse

Das Ziel dieser Arbeit ist die Entwicklung eines generischen Editors für das ProFormA-Format. In Abschnitt 3 wurde darauf eingegangen, wie das ProFormA mit den einzelnen Rollen *Aufgabenautor*, *Lehrkraft* und *Student* in Berührung kommt. Aufgabenautoren benutzen für die Erstellung von Aufgaben eigene Werkzeuge. Studenten stellen für Aufgaben nur Lösungen bereit. Als Nutzer des Editors kommen folglich nur die Lehrkräfte in Frage.

4.1. ProForma-Editor - Stärken und Schwächen

Die Auflistung der Funktionalität des ProFormA-Editors in Unterabschnitt 3.2.1 zeigt, in welchem Umfang das Editieren von ProFormA-Elementen (vgl. Unterabschnitt 2.2.1) unterstützt wird. Hier gibt es die gravierende Einschränkung, dass nur bestimmte Kombinationen von Programmiersprache und Version für eine Aufgabe zulässig sind. Ein Editor sollte keine Kombination für die verwendete Programmiersprache vorschreiben. Andere ProFormA-Elemente werden nur bedingt oder gar nicht unterstützt.

Eine Unterstützung für die Darstellung und das Editieren von Erweiterungen (*Elemente aus anderen Namensräumen*) kann sinnvoll sein. Eine Lehrkraft sollte in der Lage sein, Punkteanpassungen für eine Aufgabe durchzuführen. Punkteanpassungen werden im *grading-hints*-Element eingestellt, das nur aus Elementen anderer Namensräume besteht. Sollte ein solches Element in der Aufgabe noch nicht vorhanden sein, müssen Lehrkräfte es hinzufügen können.

Es wird schnell klar, dass der zu entwickelnde Editor eine vollständige Unterstützung *aller* Elemente des ProFormA-Formats bereitstellen muss, um Lehrkräften bei der Anpassung von Aufgaben keine Einschränkungen in den Weg zu legen. Darüber hinaus sollte das Hinzufügen, Darstellen, Editieren und Entfernen von Elementen anderer Namensräume unterstützt werden. Da der neue Editor vorab nicht wissen kann, welche Namensräume eine Aufgabe enthalten können, muss ein generischer Mechanismus konstruiert werden, mit dem sich eine Darstellung und ein Editieren von Elementen aus unbekannten Namensräumen realisieren lässt.

Der ProFormA-Editor unterstützt nützliche Funktionen wie XML-Validierung und das Abspeichern von Aufgaben als ZIP-Archiv. Die Benutzerschnittstelle ist

benutzerfreundlich. Der neue Editor sollte dies ebenfalls unterstützen.

4.2. Weitere Verbesserungen

Für Lehrkräfte, die nicht mit XML vertraut sind, kann die Fehlermeldung einer XML-Validierung kryptisch erscheinen.

Der Editor kann zusätzlich zur XML-Validierung eine weitere Prüfung aller Eingabedaten anbieten. Eingabefelder, die ungültige Werte aufweisen, können vom Editor direkt fokussiert und mit benutzerfreundlichen Fehlermeldungen versehen werden. Die generische Darstellung von XML ist für die Benutzerfreundlichkeit nicht gerade förderlich. Deshalb soll ein Pluginsystem entwickelt werden, über das sich benutzerfreundliche Schnittstellen für Elemente aus anderen Namensräumen realisieren lassen. Pluginentwickler können Plugins bereitstellen, die der Editor zur Laufzeit anbindet und für die Darstellung von Elementen nutzt.

Eine Unterstützung für eine generische Darstellung und ein Editieren von Elementen bleibt jedoch trotzdem erforderlich, da Plugins nicht immer verfügbar sein können.

Um die Benutzerfreundlichkeit einer generischen Darstellung zu maximieren, muss nach einer Möglichkeit gesucht werden, die Bearbeitung von XML in einer generischen Ansicht möglichst angenehm und verständlich zu machen. XML-Schemadefinitionen können mithilfe von *xs:annotation* und *xs:documentation* dokumentiert werden. Der Editor kann diese Dokumentationstexte zu einzelnen Elementen aus einer Schemadefinition extrahieren und bei der generischen Darstellung abrufbar machen. Das Konzept der Hilfestellung kann auf die Benutzerschnittstelle für das ProFormA-Format ausgeweitet werden. Auf diese Weise wird die Benutzerfreundlichkeit sowohl für den ProFormA-Namensraum als auch für alle anderen Namensräume erhöht.

4.3. Anwendungsfälle

Der Editor muss die folgenden grundlegenden Anwendungsfälle unterstützten können, damit Lehrkräfte Aufgaben laden, anpassen und die Ergebnisse fehlerfrei wieder abspeichern können.

Use Case 1 - Aufgabe laden

Akteure: Lehrkraft

Vorbedingung: Aufgabe wurde aus dem ZIP-Archiv entpackt und liegt als XML-Datei vor

Ereignisfluss:

1. Die Lehrkraft lädt die Aufgabe im Editor
2. Der Editor generiert die grafische Oberfläche und zeigt die Aufgabedaten an
3. Für Elemente anderer Namensräume werden Plugins geladen und in die Benutzeroberfläche eingebettet
4. Elemente anderer Namensräume, für die kein Plugin existiert, werden generisch dargestellt

Use Case 2 - Aufgabe editieren

Akteure: Lehrkraft

Vorbedingung: Aufgabe wurde im Editor geladen

Ereignisfluss:

1. Die Lehrkraft editiert Elemente aus dem ProFormA-Namensraum

Alternativen:

1.a Die Lehrkraft editiert Elemente aus einem anderen Namensraum

1. Wenn diese Elemente in der Aufgabe nicht vorhanden sind, müssen sie zu der Aufgabe hinzugefügt werden können

Use Case 3 - Editorunterstützte Validierung der Eingabedaten durchführen

Akteure: Lehrkraft

Vorbedingung: Aufgabe wurde im Editor geladen

Ereignisfluss:

1. Die Lehrkraft leitet eine Validierung aller Eingabedaten auf der Benutzerfläche ein
2. Benutzereingaben für Elemente des ProFormA-Namensraums werden auf ihre Gültigkeit untersucht
3. Für Dateien wird überprüft, ob die Dateien auch tatsächlich existieren
4. Für Elemente anderer Namensräume wird eine pluginspezifische Validierung durchgeführt, sofern Plugins für diese Elemente bereitgestellt wurden

5. Für Elemente aus den Namensraumerweiterungen, für die kein Plugin existiert, wird eine XML-Validierung durchgeführt
6. Das Ergebnis der Validierung wird der Lehrkraft angezeigt

Use Case 4 - XML-Validierung durchführen

Akteure: Lehrkraft

Vorbedingung: Aufgabe wurde im Editor geladen

Ereignisfluss:

1. Die Lehrkraft leitet eine XML-Validierung der Aufgabe ein
2. Der Editor sucht alle Schemadefinitionen zusammen, die für diese Aufgabe relevant sind
2. Es wird eine XML-Validierung der gesamten Aufgabe durchgeführt
3. Das Ergebnis der Validierung wird der Lehrkraft angezeigt

Use Case 5 - Aufgabe speichern

Akteure: Lehrkraft

Vorbedingung: Aufgabe wurde im Editor geladen

Ereignisfluss:

1. Die Lehrkraft speichert die Aufgabe im Editor ab
2. Der Editor führt eine XML-Validierung der Aufgabe durch
3. Die Aufgabe wird in der XML-Datei abgespeichert, aus der sie ursprünglich geladen wurde

Alternativen:

1.a Die Lehrkraft speichert die Aufgabe unter einem anderen Dateinamen ab

1. Die Lehrkraft gibt den neuen Dateipfad im Speicherdialog an
2. Der Editor speichert die Aufgabe unter dem neuen Dateipfad ab
3. Der Editor lädt die Aufgabe aus dem neuen Dateipfad, damit die Lehrkraft auf der soeben gespeicherten Datei weiterarbeiten kann

1.b Die Lehrkraft speichert die Aufgabe als ZIP-Archiv ab

1. Die Lehrkraft gibt den Dateipfad für das Archiv im Speicherdialog an
2. Der Editor erstellt ein ZIP-Archiv unter diesem Dateipfad
3. Die Aufgabe wird als XML-Dokument in das Archiv gelegt
4. Alle Dateien, die von der Aufgabe im *files*-Element referenziert werden, werden in das Archiv gelegt
4. Gegebenenfalls werden auch alle Schemadateien, die von der Aufgabe verwendet werden, in das Archiv gelegt (konfigurierbar)

4.4. Funktionale Anforderungen

Aus der Analyse der Stärken und Schwächen des ProFormA-Editors in Abschnitt 4.1 und den Anwendungsfällen in Abschnitt 4.3 können folgende funktionale Anforderungen an den Editor abgeleitet werden.

ANF1 - Unterstützung des ProFormA-Format
Eine vollständige Unterstützung für das Laden, Anzeigen und Editieren von Aufgaben im ProFormA-Format muss ermöglicht werden.

ANF2 - Unterstützung für andere Namensräume
Eine vollständige Unterstützung für das Laden, Anzeigen und Editieren von Elementen aus anderen Namensräumen soll dann ermöglicht werden, wenn dem Editor die entsprechenden Schemadefinitionen bereitgestellt werden.

ANF3 - Elemente aus anderen Namensräumen hinzufügen und entfernen
Der Editor soll Elemente anderer Namensräume, für die die Schemadefinitionen vorliegen, in die Aufgabe hinzufügen und aus der Aufgabe entfernen können.

ANF4 - Elemente aus anderen Namensräumen benutzerfreundlich darstellen
Plugins sollen die benutzerfreundliche Darstellung von Elementen aus anderen Namensräumen übernehmen.

ANF5 - Editorunterstützte Validierung der Eingabedaten
Der Editor soll eingegebene Daten bzgl. der ProFormA-Schemadefinition auf ihre Gültigkeit prüfen. Lehrkräfte sollen auf fehlerhafte Eingaben mit benutzerfreundlichen Fehlermeldungen hingewiesen werden.

ANF6 - XML-Validierung
Der Editor soll eine XML-Validierung von Aufgaben für alle verwendeten Schemadefinitionen durchführen können.

ANF7 - ProFormA-Aufgaben speichern
Aufgaben sollen als XML-Dokument abgespeichert werden können.

ANF8 - ProFormA-Aufgaben als ZIP-Archiv speichern
Aufgaben sollen zusammen mit allen referenzierten Dateien in einem ZIP-Archiv zusammengefügt und abgespeichert werden können.

ANF9 - Hilfestellung für das ProForma-Format anbieten
Auf der Benutzeroberfläche soll die Möglichkeit angeboten werden, zu jedem Element aus dem ProFormA-Namensraum einen Dokumentationstext über die Bedeutung des Elements abzurufen.

ANF10 - Hilfestellung für andere Namensräume anbieten
Für Elemente anderer Namensräume sollen die Dokumentationstexte aus den zugehörigen Schemadefinitionen extrahiert und bei der generischen Darstellung der Elemente angezeigt werden.

4.5. Nichtfunktionale Anforderungen

Ferner muss der Editor die folgenden Qualitätseigenschaften aufweisen.

NANF1 - Bedienbarkeit
Der Editor soll einfach zu benutzen sein. Eingabeunterstützungen sollen den Lehrkräften behilflich sein, Werte zu editieren. Die Benutzerschnittstelle soll einheitlich und konsistent aufgebaut sein.

NANF2 - Erweiterbarkeit
Neue Funktionen müssen leicht einpflegbar sein. Der Entwicklungsprozess eines Plugins soll nicht unnötig kompliziert sein.

5. Design

Dieses Kapitel stellt die entworfene Lösung für den Editor vor.

5.1. Unterstützung für ProFormA

Um eine vollständige Unterstützung für die Darstellung und das Editieren von Aufgaben im ProForma-Format erreichen zu können, bedarf es zunächst eines Werkzeugs, das den Umgang mit XML in einer Java-Anwendung ermöglicht. *Java Architecture for XML Binding* (JAXB) ist ein solches Werkzeug. JAXB bietet eine Programmierschnittstelle für Java, mit der sich XML-Dokumente an Java-Klassen binden lassen, die aus einer XML-Schemadefinition erzeugt werden ([5]). Für ProFormA werden die Java-Klassen aus der ProFormA-Schemadefinition generiert und in den Editor integriert. Das Laden, Ändern, Validieren und Speichern von ProFormA-Aufgaben läuft anschließend auf Basis dieser Klassen ab.

Für die grafische Benutzerschnittstelle wird *Java Swing* verwendet. Swing stellt eine Reihe von Komponenten zur Verfügung, mit denen sich grafische Oberflächen in einer Java-Anwendung realisieren lassen ([6]).

Im Laufe dieser Arbeit werden noch weitere Schnittstellen und Frameworks vorgestellt, die zusammen mit Swing und JAXB das Fundament bilden, auf dem der Editor seine Funktionalität für das Bearbeiten von ProFormA-Aufgaben aufbaut.

5.1.1. Aufgaben laden und speichern

Das Laden und Speichern von XML-Dokumenten wird über JAXB abgewickelt. Der Prozess nennt sich *Unmarshalling* bzw. *Marshalling*. Beim Ladevorgang überführt das Unmarshalling eine ProFormA-Aufgabe in die entsprechenden Java-Klassen, die zuvor aus der Schemadefinition generiert wurden. Analog dazu findet beim Speichervorgang des Marshalling statt, das den Prozess umkehrt und Java-Klassen in XML umwandelt. Für die Speicherung von Aufgaben als ZIP-Archiv wird das XML-Dokument nach dem Marshalling-Prozess zusammen mit allen referenzierten Dateien über Javas `ZipOutputStream`-Schnittstelle in das Archiv geschrieben.

Der Editor führt vor dem Speichervorgang eine XML-Validierung der Aufgabe durch.

5.1.2. Grafische Benutzerschnittstelle

Das ProFormA-Format soll über eine grafische Benutzerschnittstelle (kurz: GUI (Graphical User Interface)) in der Anwendung abgebildet werden. Die Verzweigtheit der ProFormA-Hauptelemente (vgl. Unterabschnitt 2.2.1) stellt eine logische Gruppierung der Elemente dar, nach der sich auch der Editor richten kann. Für die Gruppierung werden Tabs (Karteireiter) verwendet. Die einzelnen Hauptelemente werden auf Tabs abgebildet (vgl. Abbildung 5.1). Die Angaben zur Identifizierung und verwendeten Sprache einer Aufgabe aus dem Hauptelement *task* werden zusammen mit der Aufgabenbeschreibung aus dem Element *description* und den Eigenschaften zu einer Programmiersprache aus dem Element *proglang* in einem gemeinsamen Tab zusammengefasst, da sie für sich alleine genommen keine eigenständigen Tabs rechtfertigen.

Innerhalb der Tabs werden Komponenten eingebunden, die sich auf die Darstellung, das Editieren und die Validierung (vgl. Abschnitt 5.3) von konkreten Elementen spezialisieren. Bei diesen Komponenten handelt es sich um Klassen, die von Swings *JPanel* ableiten. Diese JPanel-Klassen können weitere Kind-JPanel-Klassen einbetten, die sich auf entsprechende Kindelemente spezialisieren.

Die Aufgabenbeschreibung wird durch eine HTML-Vorschau des eingegebenen Textes dargestellt. Über Buttons können Textselektionen mithilfe von HTML-Tags formatiert werden.

Listen von Kindelementen werden einheitlich über eine grafische Komponente namens *AddRemoveItemsPanel* verwaltet, über welche die Funktionalität für das Hinzufügen und Entfernen von grafischen Komponenten gehandhabt wird. Diese Komponente wird tabübergreifend für alle ProFormA-Elemente verwendet, die als Liste auftauchen können. Einheitlichkeit fördert die Bedienbarkeit.

Für viele Elemente ist die Angabe eines Wertes verpflichtend. Um Eingabefelder für solche Elemente für Lehrkräfte kenntlich zu machen, werden *Document Filters* verwendet, mithilfe derer auf fehlende oder ungültige Eingaben hingewiesen werden kann.

5.1.3. Anbindung anderer Namensräume

Im Arbeitsverzeichnis des Editors befindet sich ein Ordner *schema*, in den XSD-Dateien gelegt werden können, die vom Editor für eine generische Darstellung und ein Editieren von Elementen anderer Namensräume herangezogen werden.

Eine grafische Komponente *OtherNamespaceElementsPanel* übernimmt die Einbettung von Plugin-Editoren (vgl. Abschnitt 5.4) und generischen Editoren (vgl. Abschnitt 5.5), welche wiederum die Darstellung und das Editieren von Elementen aus anderen Namensräumen ermöglichen. Das Hinzufügen und Entfernen solcher Elemente wird ebenfalls über diese Komponente abgewickelt. Für das Hinzufügen

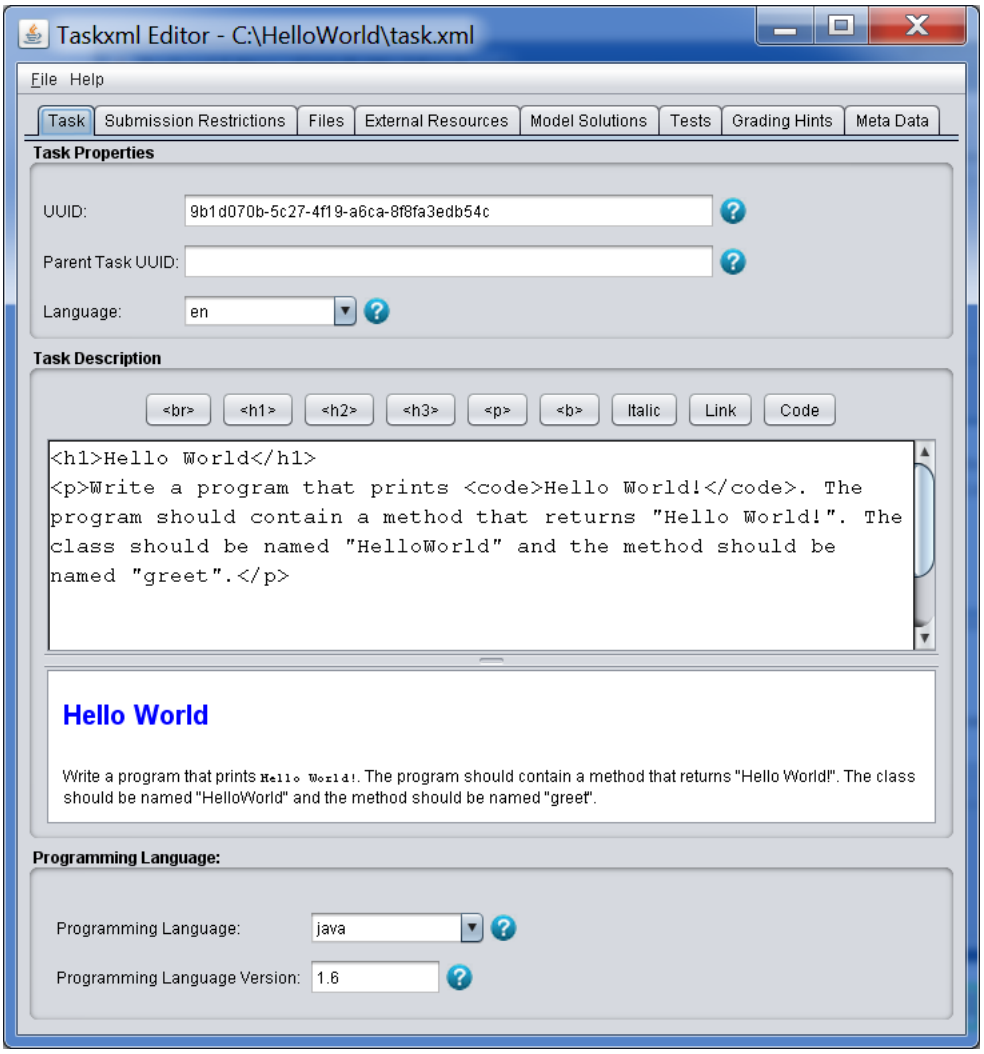

Abbildung 5.1.: Taskxml Editor: Task-Tab

muss bekannt sein, welche Elemente in anderen Namensräumen existieren. Eine solche Liste wird über die Komponente *XsdIntrospection* abgerufen (vgl. Unterabschnitt 5.5.5).

5.1.4. Hilfe

Die Hilfestellung für ProFormA-Elemente muss abrufbar sein, damit Lehrkräfte, die mit dem ProFormA-Format unvertraut sind, Dokumentationstexte zu einzelnen Elementen abrufen können. Damit die GUI nicht mit Text überlagert wird, kann zu jedem Eingabefeld ein Hilfesymbol angezeigt werden, das den Dokumentationstext nur dann anzeigt, wenn mit der Computermaus über das Symbol gefahren wird.

Die Dokumentationstexte sollen außerhalb des Editors anpassbar sein. Dazu kann Javas *ResourceBundle* verwendet werden, das auf der Verwendung von *Properties*-Dateien basiert. Über einen Schlüssel (Elementname) kann ein entsprechender Wert (Dokumentationstext) abgerufen werden. ResourceBundels haben den zusätzlichen Vorteil, dass sie eine Internationalisierung einer Anwendung ermöglichen. Abhängig von der Spracheinstellung des Computers wird die korrekte Sprachdatei geladen. Damit verbleibt nur die Aufgabe, Sprachdateien bereitzustellen.

5.2. XML-Validierung

Gemäß Anforderung ANF6 - (vgl. Abschnitt 4.4) soll der Editor eine XML-Validierung für Aufgaben unter Berücksichtigung aller Namensräume durchführen können, die in der Aufgabe verwendet werden. Auch hierfür bietet Java eine standardisierte Lösung an. Der Editor soll für die Validierung die Schnittstellen aus dem Package *javax.xml.validation* nutzen. Die Herausforderung der XML-Validierung besteht nur noch darin, die von einer Aufgabe verwendeten Schemadefinitionen zusammenzutragen, um sie an den Validierungsmechanismus weiterzureichen. Dass die ProFormA-Schemadefinition verwendet wird, ist bereits bekannt. Um festzustellen, welche Namensräume von Erweiterungselementen in einer Aufgabe verwendet werden, können die Elemente als Document Object Model-Knoten (DOM) abgerufen werden, was über die generierten JAXB-Klassen ermöglicht wird. DOM-Knoten speichern sowohl den Knotennamen als auch den Namensraum, dem der Knoten bzw. das Element angehört. Der Editor weiss auch, welche Namensräume von den Schemadefinitionen beschrieben werden, die im Ordner *schema* bereitgestellt wurden. Über einen Abgleich von Knoten-Namensraum und Schema-Namensraum ermittelt der Editor, welche Schemadefinitionen tatsächlich in der Aufgabe zum Einsatz kommen und trägt sie in einer Liste zusammen. Diese Liste wird an die Validierung weitergereicht.

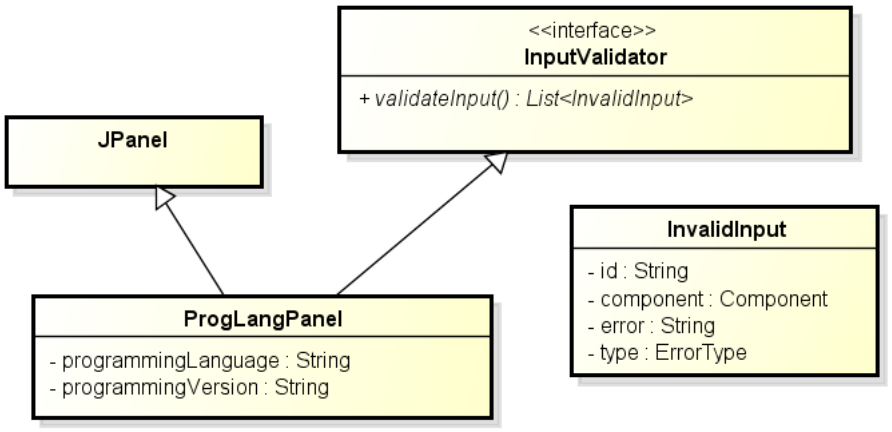

Abbildung 5.2.: Schnittstelle: InputValidator

5.3. Editorunterstützte Validierung

Spezialisierte JPanel-Klassen kümmern sich um die Darstellung und das Editieren von konkreten ProFormA-Elementen. Um die Anforderung ANF5 (vgl. Abschnitt 4.4) für eine editorunterstützte Validierung erfüllen zu können, wird eine Schnittstelle namens *InputValidator* konzipiert, die von allen JPanel-Klassen implementiert werden muss, um eine Validierung von Eingaben in diesen JPanel-Klassen ermöglichen zu können (vgl. Abbildung 5.2). Wenn Benutzer eine editorunterstützte Validierung veranlassen, wird eine *validateInput()*-Anweisung an der Wurzel der JPanel-Klassenhierarchie ausgeführt. Eine JPanel-Klasse prüft ihre Eingabemasken, erzeugt eine Liste von allen ungültigen Eingaben und reicht den *validateInput()*-Befehl anschließend in der Hierarchie weiter nach unten. Die Informationen zu ungültigen Eingaben, die von den Kindern nach Rückkehr von *validateInput()* zurückgegeben werden, werden von den Eltern in die eigene Liste mitaufgenommen, sodass am Ende der komplette Hierarchiebaum durchlaufen ist und eine Liste mit allen ungültigen Eingaben an der Wurzel vorliegt.

Der Editor wertet die Fehler im Anschluss aus. Da zu jedem Fehler eine kurze Fehlerbeschreibung und eine Referenz auf die Eingabemaske vorliegen, ist der Editor in der Lage, die entsprechende Stelle zu fokussieren und mit einem Fehlerhinweis zu versehen.

5.4. Pluginsystem

Die Anforderung ANF4 (vgl. Abschnitt 4.4) legt fest, dass Elemente aus anderen Namensräumen benutzerfreundlich unterstützt werden sollen.

Ein Plugin hat eine fest definierte Aufgabe, nämlich die Bereitstellung einer benutzerfreundlichen grafischen Oberfläche zum Darstellen und Editieren von XML-Elementen. Deshalb sollte ein Plugin auch nur Funktionen enthalten, die zur Erfüllung dieser Aufgabe beitragen. Aus diesem Grund ist die Schnittstelle eines Plugins minimal, jedoch vollständig. Der Entwicklungsprozess eines Plugins bleibt dadurch überschaubar und unkompliziert.

5.4.1. Konzept

Es sollen benutzerfreundliche Bedienoberflächen für Elemente anderer Namensräume bereitgestellt werden. Dies soll über ein Plugin-Konzept realisiert werden. Die Erstellung der Benutzeroberflächen wird auf die gleiche Weise realisiert, wie für das ProFormA-Format (vgl. Unterabschnitt 5.1.2).

Dies führt dazu, dass für unterschiedliche Elementtypen auch unterschiedliche Bedienoberflächen entstehen. Ein Plugin, dass einen Namensraum abdeckt, stellt für einzelne Elemente separate Bedienoberflächen bereit, die vom Editor angefordert und an unterschiedlichen Stellen seiner Oberfläche eingebettet werden können.

Solche Bedienoberflächen werden *Plugin-Editoren* genannt.

5.4.2. Terminologie

Im weiteren Verlauf dieser Arbeit werden neue Begriffe eingeführt, die für Verwirrung sorgen können. Deshalb werden diese Begriffe vorab definiert, um Missverständnisse zu meiden. Die Definitionen drehen sich um den Begriff *Editor*.

Der Titel dieser Abschlussarbeit lautet *Entwicklung eines generischen Editors für ein interoperables Austauschformat*. Mit *generischer Editor* ist das gesamte Programm gemeint, das die Darstellung und das Editieren von XML-Dokumenten im ProFormA-Format und in anderen Formaten auf eine *generische* Art ermöglicht. Im weiteren Verlauf wird dieses Programm als *Host-Editor* bezeichnet.

Mit der Nutzung von Plugins wird der neue Begriff *Plugin-Editor* eingeführt. Ein Plugin-Editor ist eine Komponente, die ein konkretes Element bzw. einen konkreten Elementtypen mithilfe eines spezialisierten JPanels abbildet, über das die Darstellung und das Editieren des Elements ermöglicht wird.

In Abschnitt 5.5 wird der weitere Begriff *generischer Editor* eingeführt. Beim generischen Editor handelt es sich, ähnlich wie beim Plugin-Editor, um eine Komponente, die für das Editieren von Elementen verantwortlich ist, für die kein Plugin-Editor vorhanden ist. Erfahrungsgemäß gibt es viele solcher Elemente.

Der Host-Editor *hostet* Plugin-Editoren und generische Editoren durch Einbettung in seine Benutzeroberfläche.

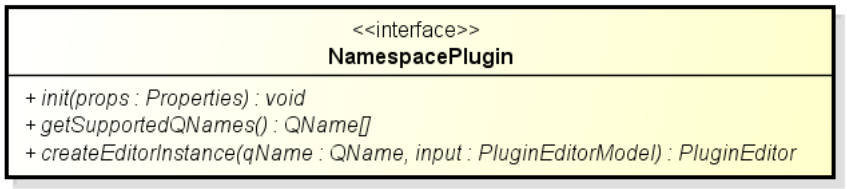

Abbildung 5.3.: Schnittstelle: NamespacePlugin

5.4.3. NamespacePlugin-Schnittstelle

Um ein Pluginsystem bereitzustellen, mit der sich Plugin-Editoren für konkrete Elementtypen erstellen lassen, muss eine Plugin-Schnittstelle konzipiert werden. Die *NamespacePlugin*-Schnittstelle (vgl. Abbildung 5.3) hat drei elementare Methoden, für die eine Beschreibung folgt.

init(props : Properties) : void

Es soll Plugins ermöglicht werden, externe Konfigurationsdateien laden zu können, sollten diese für die Funktionsweise des Plugins erforderlich sein. Als Argument erhält *init* ein Objekt vom Typ *Properties*, das vom Editor aus einer Konfigurationsdatei geladen und nach der Erstellung des Pluginobjekts an *init* übergeben wird. Dazu kann dem Plugin, das in Form einer Jar-Datei in einen *plugin*-Ordner im Arbeitsverzeichnis des Host-Editors gelegt wird, eine *.properties*-Datei beigefügt werden. die .properties-Datei muss den gleichen Namen (ohne Dateiendung) haben wie die Jar-Datei, damit sie einander zugeordnet werden können. Konfigurationsdateien für Plugins sind keine Pflicht. In diesem Fall wird ein gültiges, aber leeres Properties-Objekt an die *init*-Methode übergeben.

getSupportedQNames() : QName[]

Ein Plugin muss nach außen bekannt geben, welche Elemente es abdeckt und in welchen Namensräumen sich diese Elemente befinden. Ein *QName* steht für *Qualified Name* und identifiziert ein Element eindeutig über einen Namensraum. Ein Plugin kann Plugin-Editoren für Elemente aus unterschiedlichen Namensräumen anbieten. Der Rückgabewert dieser Methode ist ein Array von QNames, das vom Host-Editor bei der Suche nach einem geeigneten Plugin-Editor für ein Element aus einem fremden Namensraum ausgewertet wird. Sollte es vorkommen, dass zwei unterschiedliche Plugins das gleiche QName unterstützten, wird nach dem *first-come, first-served*-Prinzip gewählt.

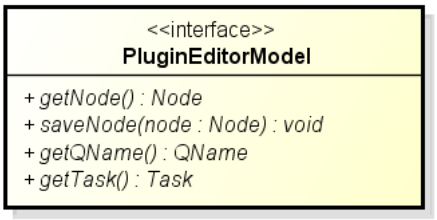

Abbildung 5.4.: Schnittstelle: PluginEditorModel

createEditorInstance(qName : QName, input : PluginEditorModel) : PluginEditor

Diese Methode ist für die Instanziierung von Plugin-Editoren verantwortlich und wird vom Host-Editor aufgerufen, wenn ein Plugin-Editor für ein vom Plugin unterstützten *QName* angefordert wird. Ein *PluginEditor* ist eine grafische Komponente, die Textfelder, Buttons usw. für das Darstellen und Editieren von Elementen (QNames) bereitstellt (vgl. Unterabschnitt 5.4.5). Ein Plugin-Editor wird mit einem Argument vom Typ *PluginEditorModel* (vgl. Unterabschnitt 5.4.4) initialisiert.

5.4.4. PluginEditorModel-Schnittstelle

Ein *PluginEditorModel* stellt die Datengrundlage dar, auf der ein Plugin-Editor (vgl. Unterabschnitt 5.4.5) arbeitet. Es folgt eine Beschreibung der PluginEditorModel-Schnittstelle.

getNode() : Node

Das Element wird in Form eines DOM-Knotens abgerufen. Ein Plugin-Editor kann entweder direkt auf dem Knoten arbeiten, oder einen Unmarshalling-Prozess auf dem Knoten ausführen, falls das Plugin JAXB verwendet.

saveNode(node : Node) : void

Speichert einen Knoten ab, damit die Änderungen für den Host-Editor sichtbar werden. Es muss sich nicht zwangsläufig um denselben Knoten (im Sinne der *Reference Equality*) handeln, der über *getNode()* abgerufen wurde. Der Host-Editor ist in der Lage zu erkennen, ob es sich dabei um den ursprünglichen oder einen neuen Knoten handelt. Knoten-Redundanzen können nicht entstehen. Der Knoten wird auch an der ursprünglichen Position im Strukturbaum abgespeichert. Diese Methode ist nur dann relevant, wenn der Plugin-Editor auf einer *Kopie* der ursprünglichen Knotendaten arbeitet, was bei der Nutzung von JAXB im Plugin beispielsweise der Fall sein könnte.

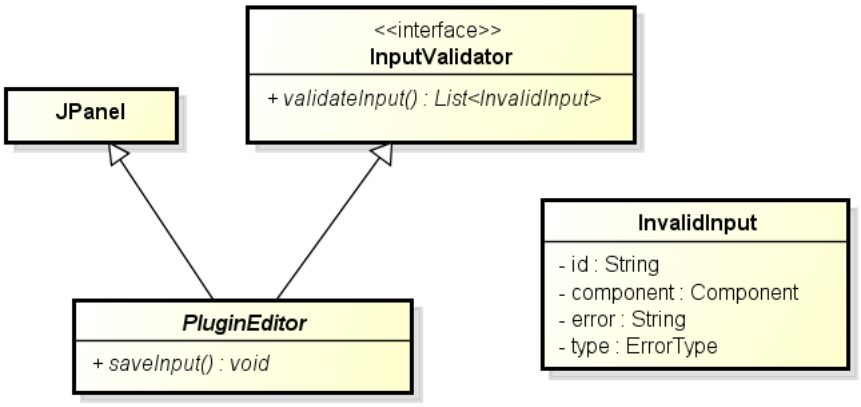

Abbildung 5.5.: Schnittstelle: PluginEditor

getQName() : QName

Ruft den QName ab, der ein Element über einem Namensraum identifiziert. Hierbei handelt es sich um den QName, für den der Plugin-Editor erstellt wurde. Es kann vorkommen, dass Elementname und Namensraum, die in einem QName gespeichert sind, für einen Plugin-Editor von Interesse sein können. In diesem Fall kann der Plugin-Editor den QName über diese Methode abrufen. Hierbei handelt es sich um eine "Bequemlichkeitsmethode", damit der Plugin-Editor den QName nicht selbst merken muss.

getTask() : Task

Ruft das Aufgaben-Objekt ab, das eine ProFormA-Aufgabe in ihrer Gesamtheit repräsentiert. Dies ist besonders dann nützlich, wenn der Plugin-Editor auf Informationen einer Aufgabe zugreifen muss, die im Element des Plugin-Editors nicht enthalten sind.

5.4.5. PluginEditor-Schnittstelle

Ein Plugin-Editor bildet ein konkretes Element über eine grafische Benutzeroberfläche ab, welche die das Darstellen und Editieren dieses Elements ermöglicht. Darüber hinaus erfüllt ein Plugin-Editor noch weitere Aufgaben. Durch die Anforderungen ANF5 (editorunterstützte Validierung, vgl. Abschnitt 4.4) wäre es sinnvoll, wenn man das in Abschnitt 5.3 vorgestellte Konzept der Datenvalidierung auch auf einen Plugin-Editor übertragen könnte. Aus diesem Grund erbt der Plugin-Editor von der Schnittstelle *InputValidator* (vgl. Abbildung 5.5). Es folgt eine Beschreibung der *PluginModel*-Schnittstelle.

saveInput() : void

Diese Methode wird vom Host-Editor aufgerufen, sobald ein Benutzer den Speichervorgang *oder* eine XML-Validierung einer Aufgabe einleitet. Ein Plugin-Editor sollte seine Daten zu diesem Zweck über die Methode *PluginEditorModel.saveNode(Node)* (vgl. Unterabschnitt 5.4.4) abspeichern, um mögliche Veränderungen an den Daten für den Host-Editor sichtbar zu machen. Dies ist nur dann relevant, wenn der Plugin-Editor auf einer Kopie des ursprünglichen Knotens gearbeitet hat, das über die *PluginEditorModel.getNode()*-Methode abgerufen wurde.

validateInput() : List⟨InvalidInput⟩

Mit dieser Methode können Änderungen, die durch Benutzer an einem Element vorgenommen wurden validiert werden. Damit kann sich ein Plugin-Editor an der editorunterstützten Validierung (vgl. Abschnitt 4.4 und 5.3) beteiligen. Dies ist ein optionaler Schritt. Falls die Validierung nicht unterstützt werden soll, muss der Wert *null* zurückgegeben werden, da eine leere Liste andernfalls die Bedeutung haben würde, dass keine Fehler aufgetreten sind.

5.4.6. Kommunikation zwischen Host-Editor und Plugin

Wenn man einen Anwendungsfall betrachtet, bei dem eine Lehrkraft eine Aufgabe lädt, in der ein Element aus einem anderen Namensraum vorkommt und es ein Plugin gibt, dass einen Plugin-Editor für dieses Element anbietet, dann kommt die folgende Kommunikation zwischen Host-Editor, Plugin und Plugin-Editor zustande (vgl. Abbildung 5.6).

1. Der Dozent lädt eine Aufgabendatei, die ein Element aus einem fremden Namensraum beinhaltet
   1.1. Der Host-Editor prüft, ob es ein Plugin gibt, das einen Plugin-Editor für das betroffene Element anbietet
   1.2. Ist dies der Fall, fordert der Host-Editor das Plugin auf, einen Plugin-Editor für dieses Element zu instanziieren und gibt dafür ein PluginEditorModel-Objekt mit, über das sich der Plugin-Editor initialisieren lässt
2. Die Lehrkraft fordert den HostEditor auf, eine editorunterstützte Validierung der Aufgabe durchzuführen
   2.1. Der Host-Editor delegiert den Befehl an den Plugin-Editor weiter und erhält eine Liste mit allen Eingabefehlern zurück, die vom Plugin-Editor gefunden wurden

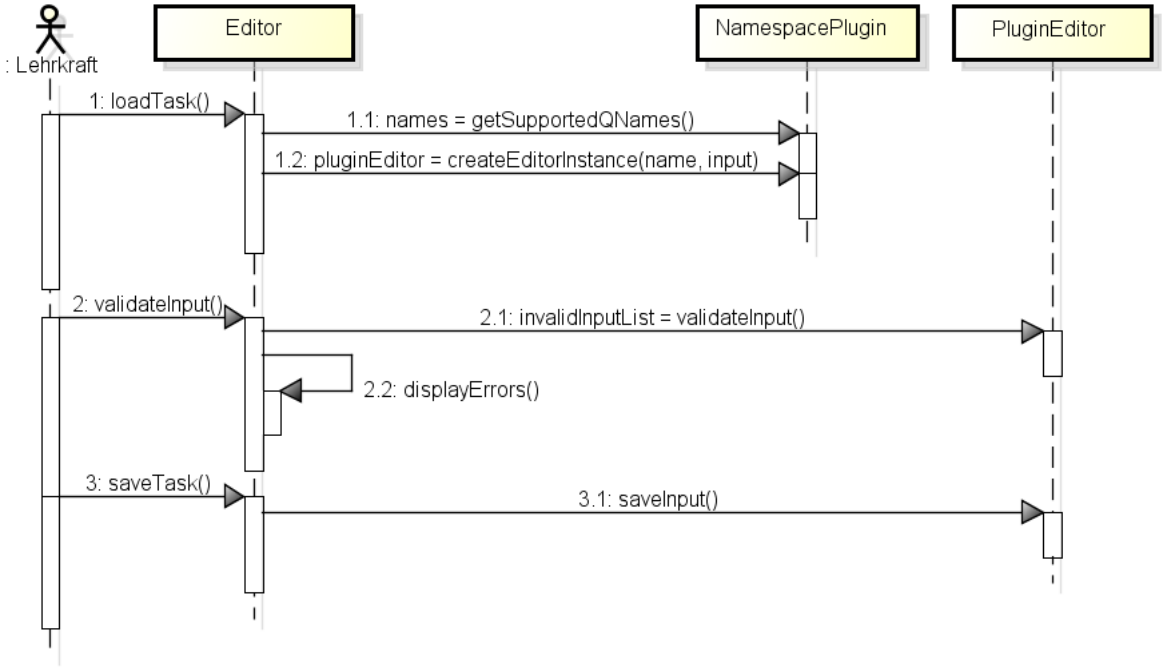

Abbildung 5.6.: Kommunikation zwischen Host-Editor und Plugin

2.2. Der Host-Editor zeigt die Eingabefehler auf der Benutzeroberfläche an

3. Die Lehrkraft veranlasst eine Speicherung der Aufgabe

   3.1. Der Host-Editor fordert den Plugin-Editor auf, seine Daten über die zuvor erhaltene Model-Schnittstelle zurück an den Ursprung zu schreiben, damit die Änderungen für den Host-Editor sichtbar werden, bevor mit der Speicherung der Aufgabe abgeschlossen werden kann

5.4.7. Plugin-Verwaltung

Für die Verwaltung von Plugins ist eine zentrale Komponente notwendig, welche die Aufgabe übernimmt, Plugins zu laden, zu instanziieren und zu initialisieren. Ferner sollen Validierungs- und Speicherbefehle für Plugin-Editoren ebenfalls über diese Komponente ablaufen. Ein *Plugin-Manager* soll diese Aufgabe übernehmen. Sämtliche Kommunikation, die zwischen Host-Editor und Plugin (inkl. Plugin-Editor) abläuft, wird über diesen Plugin-Manager abgewickelt. Die gesamte Funktionalität eines Plugin-Editors (vgl. Unterabschnitt 5.4.3) muss sich über den Plugin-Manager für alle Plugin-Editoren, die in der gesamten Anwendung im Umlauf sind, ansteuern lassen können. Dazu muss der Plugin-Manager eine Liste von allen instanziierten Plugin-Editoren mitführen. Das ist deshalb kein Problem, weil der Plugin-Manager auch die Aufgabe der Instanziierung von Plugin-Editoren übernehmen soll. Die Nutzung eines Plugin-Managers ist deshalb besonders hilfreich, weil alle Referenzen auf Plugin-Editoren an einer Stelle zentralisiert werden können.

5.4.8. Bereitstellung eines Plugins

Für die Bereitstellung eines Plugins müssen folgende Schritte durchgeführt werden.

1. Eine Klasse mit dem Namen *Plugin* implementiert die Schnittstelle *NamespacePlugin*

   a) Damit verbunden ist die Erzeugung von Klassen, welche die *PluginEditor*-Schnittstelle implementieren

2. Die Klasse *Plugin* wird im Package *de.hsh.hannover.taskxmleditor* abgelegt

3. Die Implementierungen werden in einem Java-Archiv (.jar-Datei) bereitgestellt

4. Das Archiv wird in einen Ordner *plugin* gelegt, der sich im Arbeitsverzeichnis des Editors befindet

a) Zusätzlich kann eine Konfigurationsdatei mit der Endung *.properties* beigefügt werden. Der Dateiname der Konfiguration muss mit dem Dateinamen der Jar-Datei (ohne Dateiendung) übereinstimmen.

Damit ist die Bereitstellung abgeschlossen. Der Editor prüft beim Programmstart, ob im *plugin*-Ordner Jar-Dateien vorliegen, in denen sich eine Klasse *de.hsh.hannover.taskxmleditor.Plugin* befindet. Ist dies der Fall, lädt der Editor das Plugin und die zugehörige Konfigurationsdatei, sofern eine vorhanden ist. Anschließend wird das Plugin initialisiert und gegebenenfalls für Elemente aus anderen Namensräumen verwendet.

5.5. Unterstützung für andere Namensräume

Gemäß Anforderung ANF2 (vgl. Abschnitt 4.4) muss der Editor in der Lage sein, Elemente aus anderen Namensräumen darzustellen und bearbeiten zu können. Es muss eine generische Editor-Komponente entwickelt werden, die diesen Bedarf abdeckt. Eingesetzt soll diese Komponente immer dann, wenn Elemente aus fremden Namensräumen nicht von Plugins unterstützt werden können, oder wenn die Instanziierung von einem Plugin-Editor aufgrund eines Programmierfehlers fehlschlägt.

Ein Editor, der DOM-Knoten auf eine generische Weise darstellen *und* editieren können muss, steht einigen Herausforderungen gegenüber, die es zu bewältigen gilt.

5.5.1. Darstellung

Die Struktur eines Knotens muss visuell dargestellt werden. Knoten können selbst wiederum Knoten enthalten, die der Editor rekursiv darstellen muss. Da Knoten von der Strukturlänge und -tiefe sehr groß werden können, sollten Benutzer in der Lage sein, einzelne Strukturteile des Knotens ein- und ausblenden zu können, damit sie sich selektiv auf den für sie relevanten Strukturteil konzentrieren können.

Die Baumstruktur kann am besten über eine tabellenartige Baumansicht repräsentiert werden, die das Auf- und Zuklappen von Knotenteilen ermöglicht und den Knotenwert sowie eine Hilfestellung in zusätzlichen Spalten anzeigt (vgl. Abbildung 5.8).

5.5.2. Editieren

Die Darstellung von Knotenstrukturen ist nützlich, reicht aber noch nicht aus. Benutzer müssen in der Lage sein, neue Knoten in den Strukturbaum einpflegen

und auch löschen zu können. Weder die XML-Schemadefinition des ProFormA-Formats, noch die Schemadefinitionen der Namensraumerweiterungen lassen eine willkürliche Strukturänderung zu. Der Editor muss deshalb dort Änderungen an der Struktur erlauben, wo es laut Schemadefinition möglich ist und dort unterbinden, wo es aufgrund des *momentanen Zustands* der Struktur untersagt ist oder grundsätzlich nicht möglich ist. Auch muss die Tatsache berücksichtigt werden, dass Schemadefinitionen abhängig von der Position in der XML-Struktur unterschiedlichen Inhalt erwarten. Nicht nur das Editieren der Knotenstruktur muss möglich sein, sondern auch das Editieren der Knotenwerte. Hier schlägt ebenfalls das gleiche Prinzip wie bei der Strukturänderung an. Nicht alle Knotenwerte können änderbar sein. Manche Knoten erlauben nur Werte, die aus einer vordefinierten Menge zugelassen sind (*Enumerationen*), andere legen bei Zahlenwerten eine maximale Ober- und Untergrenze fest. Als Grundsatz gilt für das Editieren: ob Struktur und Wert eines Knotens veränderbar sind, ist in der dazugehörigen Schemadefinition beschrieben. Ein generischer Editor muss in der Lage sein, den Regeln eines solchen Schemas Folge leisten zu können, damit der Editor am Ende gültige XML-Dokumente produziert.

5.5.3. Eingabeunterstützungen

Der Editor muss Benutzer bei der Eingabe von Daten unterstützen können. Im Falle der Strukturänderung soll einem Benutzer beim Anfügen eines neuen Knotens alle möglichen Elemente und Attribute angezeigt werden, die an dieser Position erlaubt sind. Faktoren wie *minOccurs* und *maxOccurs* müssen dabei berücksichtigt werden. Für das Editieren des Knotenwertes soll der zugrundeliegende Datentyp des Knotens inspiziert und ein entsprechender Datentypeditor angezeigt werden. Beispielsweise kann für eine *Enumeration* eine *Dropdown-Liste* verwendet werden und für den Datentyp *Boolean* eine *CheckBox*.

5.5.4. Hilfe

Anforderung ANF10 (vgl. Abschnitt 4.4) legt fest, dass auch für Elemente aus fremden Namensräumen eine Hilfestellung unterstützt werden muss. Hilfestellungen sind für Elemente, die generisch gehandhabt werden, besonders hilfreich, weil hier nach jeder Möglichkeit gesucht werden muss, eine generische Darstellung bzgl. der Bedienbarkeit zu optimieren (vgl. Anforderung NANF1 in Abschnitt 4.5).

Da der Editor vorab nicht wissen kann, welche Namensräume zusätzlich zum ProFormA-Namensraum in einer Aufgabe vorkommen können, kann eine Hilfestellung, wie sie in Unterabschnitt 5.1.4 vorgestellt wird, nicht im Voraus als fertiges Textpaket angeboten werden. Allerdings kann der Editor die Möglichkeit

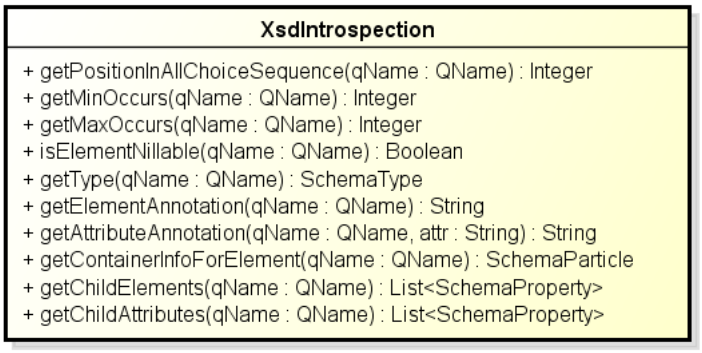

Abbildung 5.7.: Klasse: XsdIntrospection

der *Bereitstellung* von Hilfstexten zu Elementen aus fremden Namensräumen unterstützen. Autoren von Schemadefinitionen können ermutigt werden, Hilfstexte für solche Elemente bereitzustellen, die der Editor aus dem Schema extrahieren und für Benutzer zu den Elementen anzeigen kann.

Ermöglicht wird die Bereitstellung von Hilfstexten durch *xs:annotation* und *xs:documentation*, die innerhalb von Schemadefinition in die betroffenen Element eingebettet werden (vgl. Listing 5.1). Der Editor extrahiert den Hilfstext für ein Element aus der Schemadefinition und zeigt sie in der generischen Darstellung von Elementen mit an (vgl. Abbildung 5.8).

Listing 5.1: Dokumentation für ein Element

```
1 <xs:element name="foo" type="tns:bar">
2   <xs:annotation>
3     <xs:documentation>Help text goes here.</xs:documentation>
4   </xs:annotation>
5 </xs:element>
```

5.5.5. XSD-Introspektion

Um die Darstellung, das Editieren, die Eingabeunterstützungen und die Anzeige von Hilfstexten zu ermöglichen, wurde ein Mechanismus zur Untersuchung und Auswertung von XML-Schemadefinitionen ausgearbeitet. Der Mechanismus basiert auf dem Konzept der *Typ-Introspektion*, die es einer Anwendung erlaubt, Informationen über einen Datentyp zur Laufzeit zu untersuchen. Für die Ermöglichung der Typ-Introspektion für XML-Elemente (*XSD-Introspektion*) macht sich der Editor das Framework *XMLBeans* zunutze ([8]).

Für die Arbeit mit *XMLBeans* wird eine Klasse namens *XsdIntrospection* erstellt (vgl. Abbildung 5.7), die nach dem *Facade*-Entwurfsmuster entworfen ist. In

der Klasse wird die wichtigste Funktionalität zusammengetragen, die für den generischen Editor relevant ist. Informationsabfragen zu einem XML-Element werden über *XsdIntrospection* abgewickelt.

Die Funktionsweise von *XsdIntrospection* wird in Unterabschnitt 6.2.1 vertieft.

5.5.6. Editieren von Knotenstrukturen

Für die Strukturmanipulation soll die generische Editor-Komponente eine Schnittstelle für den Benutzer anbieten, mit der sich neue Knoten in eine Struktur einfügen oder Knoten aus einer Struktur entfernen lassen können. Die Darstellung von Knotenstrukturen wird über eine Baumansicht realisiert (vgl. Abbildung 5.8). Als Benutzerschnittstelle soll für jeden Knoten ein individuelles Kontextmenü generiert werden. Das Kontextmenü soll Optionen zum Einfügen von neuen Kindknoten an den jeweiligen Positionen anbieten. Auch das Entfernen von Knoten soll angeboten werden. Allerdings darf dies nicht willkürlich passieren. Das Kontextmenü muss den aktuellen Zustand des selektierten Knotens berücksichtigen und seinen Datentypen auswerten. Hierzu werden Informationen über *minOccurs* und *maxOccurs* des betroffenen Knotentyps über die *XsdIntrospection*-Klasse abgerufen. Bei Attributen muss die *required*-Eigenschaft berücksichtigt werden. Das Ergebnis ist ein intelligentes Kontextmenü, das einem Benutzer anzeigt, welche Elemente und Attribute noch zum selektierten Knoten hinzugefügt werden dürfen und ob der Knoten selbst gelöscht werden darf. Wenn man von einem Szenario ausgeht, in dem ein Benutzer ein neues Element in eine bestehende Knotenstruktur einfügen möchte, lässt sich der Ablauf wie folgt beschreiben:

1. Der Benutzer leitet die Erzeugung eines Kontextmenüs durch einen Rechtsklick der Maus auf einem Knoten ein.

2. Der Editor prüft nach, ob es sich bei dem Knoten um ein Attribut oder ein Element handelt.

    a) Für Attribute können keine Kindelemente hinzugefügt werden.

    b) Für Elemente ruft der Editor über *XsdIntrospection* ab, welche Knotentypen (sowohl Elemente als auch Attribute) in diesem Elternelement gemäß Schemadefinition enthalten sein dürfen. Jeder potentielle Kindknoten wird iterativ untersucht.

        i. Handel es sich bei dem Kindknoten um ein Attribut, so muss nur geprüft werden, ob das Attribut bereits in der Knotenstruktur verfügbar ist, da in einem Element maximal ein Attribut eines bestimmten Typs enthalten sein darf.

   ii. Für Elementtypen wird geprüft, wie oft dieser Elementtyp im Elternknoten enthalten sein darf (*min-* und *maxOccurs*). Diese Informationen werden mit dem aktuellen Zustand der Knotenstruktur des Elternknotens abgeglichen.

3. Alle Elemente und Attribute, die an den in Schritt 1 selektierten Knoten angefügt werden dürfen, werden im Kontextmenü angezeigt, wobei sie in Untermenüs nach *Element* und *Attribut* gruppiert werden.

4. Der Benutzer entscheidet sich für das Hinzufügen eines neuen Elementknotens und wählt die entprechende Option aus.

5. Das gewünschte Element wird konstruiert. Da das Element selbst wiederum Kindelemente und -attribute enthalten kann, die laut Schemadefinition im Element enthalten sein *müssen* (z. B. *required*-Attribute), erzeugt der Editor für das gewünschte Element automatisch eine vorkonfigurierte Knotenstruktur, die alle Kindknoten enthält, die für dieses Element verpflichtend sind. Dieser Prozess wird rekursiv auf die Kindknoten angewandt, da dort auch das gleiche Prinzip gilt. Auf diese Weise entsteht eine gültige Knotenstruktur, die sich problemlos gegen ein Schema validieren lässt.

6. Die neu erstellte Struktur wird als Kindknoten zum Elternknoten hinzugefügt, der vom Benutzer in Schritt 1 selektiert wurde.

7. Die Ansicht wird aktualisiert, der neue Knoten erscheint an der entsprechenden Position.

Das Entfernen von Knoten funktioniert nach einem ähnlichen Prinzip, ist jedoch deutlich simpler, da hier nur die *required*-Eigenschaft von Attributen und die *minOccurs*-Eigenschaft von Elementen berücksichtigt werden muss.

5.5.7. Editieren von Knotenwerten

Knoten können Werte enthalten. Zu jedem Knoten wird der entsprechende Knotenwert in einer Spalte *Value* angezeigt (vgl. Abbildung 5.8). Mithilfe der *XsdIntrospection*-Klasse kann der zugrundeliegende Datentyp eines Knotens abgerufen werden. Abhängig vom Typ wird beim Selektieren eines Knotenwertes in der Value-Spalte ein Editor angezeigt, der das Editieren dieses Knotenwertes vereinfacht. Für *Enumerationen* wird beispielsweise eine *Dropdown-Liste* verwendet.

5.5.8. Editieren ohne Schemadefinition

Eine hilfreiche Unterstützung für das Editieren von Knoten wird durch die Bereitstellung von Schemadefinitionen ermöglicht. Sind diese Definitionen nicht vorhanden, kann der generische Editor keine vernünftige Unterstützung mehr anbieten. Nichtsdestotrotz ist noch nicht alles verloren. Es wird nicht möglich sein, neue Knoten an eine existierende Knotenstruktur anzufügen, da dies sonst schnell zu XML-Dokumenten führen kann, die sich nicht mehr erfolgreich gegen eine Schemadefinition validieren lassen. Die Option, neue Knoten anzufügen, wird in diesem Fall deshalb abgestellt. Ein DOM-Knoten, der aus einem XML-Dokument geladen wurde, kann aber weiterhin über eine Baumansicht dargestellt werden. Das Editieren von Knotenwerten ist trotz fehlender Schemadefinition prinzipiell auch möglich. Eine Gefahr von falschen Benutzereingaben besteht jedoch auch hier.

5.5.9. Nutzung der Plugin-Infrastruktur

Der generische Editor erfüllt die gleiche Aufgabe wie ein Plugin-Editor: das Darstellen und Editieren von Elementen aus anderen Namensräumen. Damit der generische Editor die Vorteile der Plugin-Infrastruktur (vgl. Unterabschnitt 5.4.7) nutzen kann, erbt der generische Editor von der *PluginEditor*-Schnittstelle (vgl. Abbildung 5.5). Damit können die Vorteile der Zentralisierung von Komponenten für die Verwaltung von Elementen aus anderen Namensräumen auch für den generischen Editor in Anspruch genommen werden. Die Verwaltungsaufgaben von generischen Editor-Instanzen werden somit an die existierende Infrastruktur des Plugin-Managers abgedrückt.

5.6. Ein Vergleich der Editor-Komponenten

Der Plugin-Editor und der generische Editor stellen Elemente aus anderen Namensräumen dar. Deshalb ist die Aufstellung eines Vergleichs der beiden Editortypen besonders spannend. Verglichen werden dabei *Benutzerfreundlichkeit*, *Verfügbarkeit* und *Datenvalidierung*.

5.6.1. Benutzerfreundlichkeit

Die Benutzerfreundlichkeit lässt sich am besten anhand eines visuellen Beispiels der Benutzerschnittstelle vergleichen. Dazu wird ein Element aus einem fremden Namensraum betrachtet. Die Editoren sollen das Element *test-group* darstellen, das die Punktzahlen für automatisierte Prüfungen von Aufgaben festlegt.

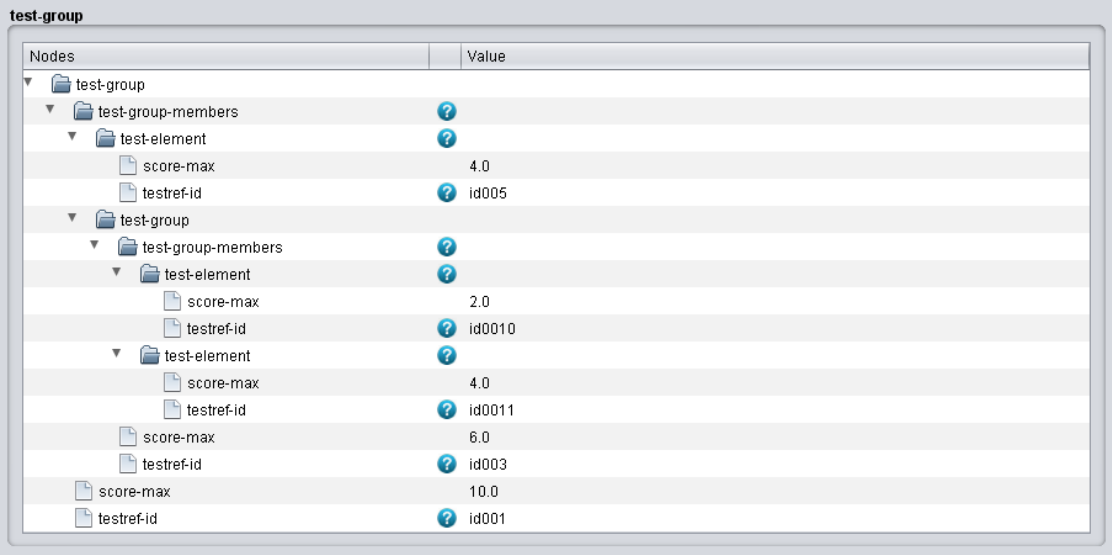

Abbildung 5.8.: Generische Editor-Komponente

Generischer Editor

In Abbildung 5.8 ist eine Tabelle mit aufklappbaren Zeilen zu sehen. Während sich die baumartige Struktur von XML-Dokumenten hervorragend über eine Baumansicht abbilden lässt, ist es für die Benutzerfreundlichkeit nicht optimal, wenn alle Knoten auf die gleiche Weise dargestellt werden. Die Relevanz von besonders wichtigen Knoten kann dadurch nicht hervorgehoben werden.

Die Knotennamen stammen direkt aus der Schemadefinition. Da Schemaautoren bei der Namensgebung für Elemente oftmals kryptische Namen verwenden, kann es vorkommen, dass Benutzer mit den Knotennamen nicht viel anfangen können. Eine Hilfestellung durch ein Hilfesymbol ist bei der generischen Darstellung deshalb ein willkommener Pluspunkt.

Knotenwerte werden durch verschiedene Eingabeeditoren vorgenommen, die in Abhängigkeit vom Datentyp des Knotenwertes anspringen. Neue Knoten können über ein Kontextmenü angefügt und entfernt werden.

Plugin-Editor

In Abbildung 5.9 ist das Element über einen spezialisierten Plugin-Editor dargestellt. Die baumartige Struktur des Knotens wird durch eine Gruppierung von GUI-Elementen realisiert. Gruppen werden nach dem Inhalt eines auserwählten Knotenwertes (Testreferenz) benannt, was die Übersichtlichkeit der Gruppierung fördert. Das Kontextmenü wurde durch spezielle Knöpfe für das Hinzufügen und Entfernen der jeweiligen Knoten ersetzt. Eine Hilfestellung über ein Hilfesymbol ist hier ebenfalls an den entsprechenden Stellen angebracht.

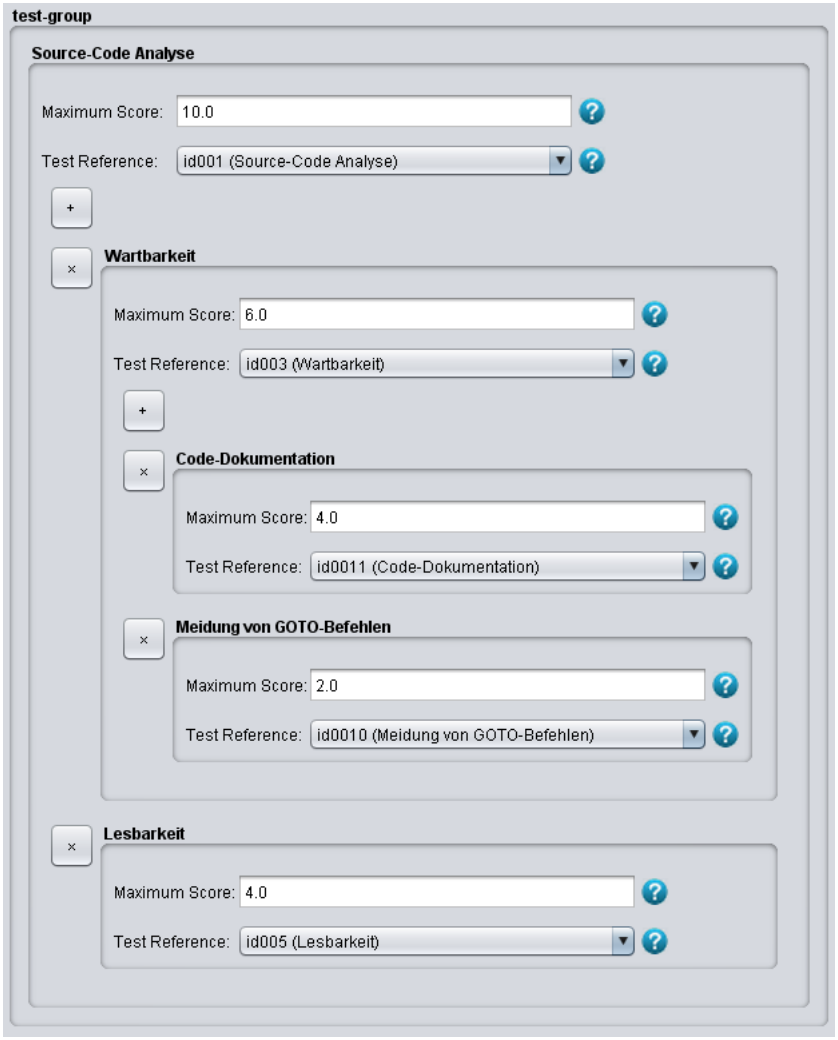

Abbildung 5.9.: Handgefertigte Plugin-Darstellung

Fazit

Pluginentwicklern steht es frei zu entscheiden, wie die Benutzerschnittstelle für Elemente gestaltet werden soll. Hier lässt sich durch eine intelligente Anordnung und Gruppierung einzelner GUI-Elemente viel herausholen. Zusätzlich können kontextuell relevante Informationen aus anderen Quellen herangezogen und angezeigt werden. Bei dem generischen Editor ist die Darstellung von Informationen auf das beschränkt, was in einem Knoten und seinen Kindknoten gespeichert ist.

5.6.2. Verfügbarkeit

Elemente aus anderen Namensräumen müssen jederzeit darstellbar sein. Auch muss die Tatsache berücksichtigt werden, dass sich Schemadefinitionen für Elemente verändern können.

Generischer Editor

Durch die generische Arbeitsweise kann der generische Editor beliebige Elemente aus beliebigen Namensräumen jederzeit darstellen, auch wenn keine Schemadefinition zu diesen Elementen vorliegt. Liegt ein Schema vor, so können Elemente auch editiert werden.

Für den generischen Editor spielt es keine Rolle, dass sich Schemadefinitionen ändern können, da ein Schema zur Laufzeit ausgewertet wird.

Plugin-Editor

Ein Plugin-Editor ist auf die Darstellung eines bestimmten Elements spezialisiert. Verfügbar sind Plugin-Editoren folglich nur für Elemente, für die sie auch entwickelt wurden.

Wenn sich eine Schemadefinition ändert, ändert sich (wahrscheinlich) auch der Uniform Resource Identifier (URI) des Namensraums, für das sie Elemente definiert. Die Änderung der URI soll eine Änderung an der Definition reflektieren. Das ist dann der Fall, wenn eine Namensraum-URI als Form der Versionierung für eine Schemadefinition verwendet wird. Der Plugin-Editor identifiziert sein Element durch einen *QName*, der aus Namensraum-URI und Elementname besteht. Sobald die Namensraum-URIs nicht mehr miteinander übereinstimmen, kann der Plugin-Editor nicht für das Element verwendet werden, auch wenn sich dieses Element selbst nicht in der Schemadefinition verändert hat. Dies ist ein notwendiges Übel, da es sonst keine Möglichkeit gibt, festzustellen, in welcher *Version* ein Element in einem XML-Dokument vorliegt.

Fazit

Bezüglich Verfügbarkeit kann man sich darauf verlassen, dass der generischen Editor Elemente zu jeder Zeit darstellen und editieren kann, wenn das entsprechende Schema vorliegt. Von einem Plugin-Editor kann ein Element nur dann dargestellt werden, wenn ein Plugin-Editor bereitgestellt wurde und die *QNames* übereinstimmen

5.6.3. Datenvalidierung

Bei der Eingabe von Daten können Fehler entstehen. Es wird verglichen, wie die Editoren jeweils mit falschen Dateneingaben umgehen.

Generischer Editor

Der generische Editor setzt die Einschränkungen um, die im Schema für Attribute und Elementwerte vorgegeben werden. Dazu wird eine XML-Validierung des dargestellten Elements durchgeführt.

Plugin-Editor

Der Plugin-Editor *kann* eine editorunterstützte Validierung bereitstellen, die zusätzlich zur XML-Validierung durchgeführt wird.

Pluginentwickler können diese Möglichkeit nutzen und weitere inhaltliche Prüfungen einbauen, die über eine Schemadefinition nicht umgesetzt werden können. Wenn man das Beispiel aus Abbildung 5.9 betrachtet, sieht man, dass die Punktezahl eines Elternelements *test-group* die Summe der einzelnen Punktezahlen von Kindelementen bildet. Hierfür kann eine Prüfung eingebaut werden, damit nicht versehentlich die falsche Summe in ein Elternelement eingetragen wird.

Tatsächlich könnte man hier noch einen Schritt weiter gehen und das Plugin so implementieren, dass die Summe der Punktezahl eines Elternelements automatisch aus der Eingabe der Punktezahlen für die Kindelemente ermittelt wird.

Fazit

Bei der Pluginentwicklung ist eine Bereitstellung der editorunterstützten Datenvalidierung zwar mit zusätzlichem Aufwand verbunden, dafür können Eingabefehler auf ein Minimum reduziert werden, wenn man zusätzliche Prüfungen einbaut, die eine XML-Validierung nicht hergeben kann.

6. Implementierung

In diesem Kapitel werden die wichtigen Implementierungen der in Kapitel 5 ausgearbeiteten Mechanismen vorgestellt.

6.1. Plugin

Im Laufe dieser Arbeit wurde ein Plugin entwickelt, dass eine benutzerfreundliche Benutzeroberfläche für Elemente aus dem Namensraum *urn:grappa:tests:hierarchical:v0.0.1* zur Verfügung stellt. In diesem Abschnitt wird die Implementierung für das Element *computing-resources* demonstriert. Das *computing-resources*-Element legt Werte fest, die den Betriebsmittelverbrauch von Gradern steuern, die bei einem automatisierten Bewertungsprozess von studentischen Lösungseinreichungen zu einer Aufgabe anfallen ([3]).

Die Schemadefinition von *computing-resources* ist in Listing 6.1 aufgeführt.

Listing 6.1: Schemadefinition: computing-resources

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3     xmlns:grp="urn:grappa:tests:hierarchical:v0.0.1"
          targetNamespace="urn:grappa:tests:hierarchical:v0.0.1"
4     elementFormDefault="qualified" attributeFormDefault="
          unqualified">
5     <xs:element name="computing-resources" type="grp:computing-
          resources-type" />
6     <xs:complexType name="computing-resources-type">
7         <xs:all>
8             <xs:element name="max-runtime-seconds-wallclock-time"
9                 type="xs:int" nillable="true" />
10            <xs:element name="max-disc-quota-kib" type="xs:int"
11                nillable="true" />
12            <xs:element name="max-mem-mib" type="xs:int" nillable="
                  true" />
13        </xs:all>
14    </xs:complexType>
15 </xs:schema>
```

Um die Listings überschaubar zu halten, wurde die Plugin-Unterstützung für das *computing-resources*-Element auf die Eigenschaft *max-runtime-seconds-wallclock-*

time reduziert. Die Implementierungen für *max-disc-quota-kib* und *max-mem-mib* funktioniert nach dem gleichen Prinzip wie für *max-runtime-seconds-wallclock-time*.

Für den Umgang mit XML wird in dem Plugin das JAXB-Framework verwendet. Dazu wurde aus der Schemadefinition für *computing-resources* eine Java-Klasse *ComputingResourcesType* mithilfe eines entsprechenden Tools (IntelliJ) generiert, die in Listing 6.2 zu sehen ist.

Listing 6.2: ComputingResourcesType

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "computing-resources-type", propOrder = {
3 })
4 public class ComputingResourcesType {
5   @XmlElement(name = "max-runtime-seconds-wallclock-time",
        required = true, type = Integer.class, nillable = true)
6   protected Integer maxRuntimeSecondsWallclockTime;
7
8   public Integer getMaxRuntimeSecondsWallclockTime() {
9     return maxRuntimeSecondsWallclockTime;
10  }
11
12  public void setMaxRuntimeSecondsWallclockTime(Integer value) {
13    this.maxRuntimeSecondsWallclockTime = value;
14  }
15 }
```

Auf dieser Basis kann nun ein Plugin-Editor (vgl. Unterabschnitt 5.4.5) erzeugt werden.

6.1.1. PluginEditor

Der Plugin-Editor, der von *JPanel* erbt, stellt die grafische Oberfläche zur Verfügung, die für das Editieren des *maxRuntimeSecondsWallclockTime*-Feldes einer *ComputingResourcesType*-Instanz vorgesehen ist. Listing 6.3 zeigt, wie die Oberfläche aufgebaut wird und wie ein *JTextField* über einen *DocumentListener* Benutzereingaben in das *maxRuntimeSecondsWallclockTime*-Feld überträgt.

Die Anweisung *Str.get("CR.MaxRuntime.Help")* ruft den Hilfstext für den Schlüssel *CR.MaxRuntime.Help* aus einer Sprachdatei ab (vgl. Unterabschnitt 5.1.4). Über den *Gui.combineWithHelp()*-Aufruf wird neben dem *maxRuntimeField-JTextField* ein Fragezeichensymbol platziert, über das der Hilfstext angezeigt werden kann.

Listing 6.3: ComputingResourcesEditor: Felder und Konstruktor

```
1 public class ComputingResourcesEditor extends PluginEditor {
2     private PluginEditorModel model;
```

```
3     private ComputingResourcesType data;
4     private JTextField maxRuntimeField;
5
6     public ComputingResourcesEditor(PluginEditorModel model)
          throws Exception {
7         this.model = model;
8         maxRuntimeField= new JTextField();
9         setBorder(BorderFactory.createTitledBorder("Computing
              Resources"));
10        setLayout(new GridBagLayout());
11        add(new JLabel(Str.get("CR.MaxRuntime")), Gui.createGbc(0,
              0));
12        add(Gui.combineWithHelp(maxRuntimeField, Str.get("CR.
              MaxRuntime.Help")), Gui.createGbc(1, 0));
13
14        data = PluginHelper.unmarshal(ComputingResourcesType.class
              , model.getNode());
15
16            maxRuntimeField.getDocument().addDocumentListener(new
                  DocumentListenerAdapter() {
17            @Override
18            public void contentChanged() {
19                if (!maxRuntimeField.getText().isEmpty())
20                    data.setMaxRuntimeSecondsWallclockTime(Integer
                          .parseInt(maxRuntimeField.getText()));
21                else
22                    data.setMaxRuntimeSecondsWallclockTime(null);
23            }
24        });
25        new PositiveIntegerFilter(maxRuntimeField, true);
26    }
```

Bei dem Aufruf *PluginHelper.unmarshal()* handelt es sich um eine "Bequemlichkeitsmethode", die eine *JAXBContext*-Instanz erstellt und den Knoten aus *PluginEditorModel.getNode()* in ein Objekt des Typs *ComputingResourcesType* umwandelt. Analog dazu gibt es auch eine *PluginHelper.marshal()*-Methode, wie in der Methode *saveInput()* in Listing 6.4 zu sehen ist. Dort wird die *ComputingResourcesType*-Instanz wieder zurück in einen Knoten umgewandelt, der schließlich an *PluginEditorModel.saveNode()* übergeben werden kann.

Listing 6.4: ComputingResourcesEditor: saveInput()

```
1     @Override
2     public void saveInput() throws Exception {
3         Node n = PluginHelper.marshal(ComputingResourcesType.class
              , model.getQName(), data);
4         model.saveNode(n);
5     }
```

Eine Implementierung der Methode *validateInput()* wird in Listing 6.5 aufgezeigt. Aufgrund der *PositiveIntegerFilter*-Instanz (vgl. Listing 6.3), die sich als *DocumentFilter* im *maxRuntimeField-JTextField* registriert und nur Eingaben von positiven ganzen Zahlen zulässt, ist die Prüfung des *maxRuntimeField*-Wertes in *validateInput()* nicht notwendig, wird hier aber zu Demonstrationszwecken aufgeführt.

Listing 6.5: ComputingResourcesEditor: validateInput()

```
1     @Override
2     public List<InvalidInput> validateInput() {
3         ArrayList<InvalidInput> a = new ArrayList<>();
4         if(!org.apache.commons.lang3.StringUtils.isNumeric(
              maxRuntimeField.getText()))
5             a.add(new InvalidInput("MaxRuntime", maxRuntimeField,
                  "Value must be a positive integer"));
6
7         return a;
8     }
9 }
```

6.1.2. NamespacePlugin

Um den Plugin-Editor nach außen verfügbar zu machen, wird in Listing 6.6 eine Plugin-Klasse erstellt, welche die *NamespacePlugin*-Schnittstelle (vgl. Unterabschnitt 5.4.3) implementiert.

Das Feld *computingResourcesQn* definiert den *QName*, für den der Plugin-Editor (vgl. Unterabschnitt 6.1.1) gelten soll. *getSupportedQNames()* gibt den *QName* nach außen bekannt während *createEditorInstance()* die eigentliche *PluginEditor*-Instanz erzeugt.

Listing 6.6: NamespacePlugin

```
1 public class Plugin implements NamespacePlugin {
2   private QName computingResourcesQn =
3             new QName("urn:grappa:tests:hierarchical:v0.0.1",
4                       "computing-resources");
5
6   @Override
7   public void init(Properties props) { }
8
9   @Override
10  public QName[] getSupportedQNames() {
11    return new QName[]{ computingResourcesQn };
12  }
13
14  @Override
```

```
15    public PluginEditor createEditorInstance(QName qName,
          PluginEditorModel inputData) throws Exception {
16      if(computingResourcesQn.equals(qName))
17        return new ComputingResourcesEditor(inputData);
18      return null;
19    }
20  }
```

6.2. Generischer Editor

Die Kernfunktion des generischen Editors ist die XSD-Introspektion (vgl. Unterabschnitt 5.5.5), ohne die eine sinnvolle Unterstützung für XML-Elemente aus Schemadefinitionen nicht möglich gewesen wäre. Deshalb fokussiert sich dieser Abschnitt auf die Implementierung der *XsdIntrospection*-Klasse.

6.2.1. XSD-Introspektion

Ermöglicht wird die XSD-Introspektion durch das *XMLBeans-Framework*, das eine Reihe unterschiedlicher Werkzeuge für die Verwendung von XML in Java anbietet. Interessant ist dabei vorallem, dass sich über eine Reihe von Klassen das Schema-Objektmodel einer zugrundeliegenden Schemadefinition abrufen lässt ([8]). Die *XsdIntrospection*-Klasse (vgl. Abschnitt 5.7) macht sich dieser Klassen zunutze.

Bevor auf die Implementierung der *XsdIntrospection*-Klasse eingegangen wird, sollten die verwendeten Klassen von XMLBeans einmal kurz erläutert werden.

SchemaTypeSystem Das *SchemaTypeSystem* repräsentiert eine Menge von XML-Schemadefinitionen unterschiedlicher Namensräume. Über das *SchemaTypeSystem* lassen sich alle Informationen zu Elementen, Attributen, komplexen Typdefinitionen usw. abrufen.

SchemaType Jedes Schemaobjekt einer Schemadefinition wird durch einen *SchemaType* repräsentiert. Der *SchemaType* gibt neben grundsätzlichen Metainformationen Auskunft darüber, um was für eine Art von Schemaobjekt es sich dabei handelt. Beispielsweise könnte es eine Attributdefinition oder ein Element vom Typ *simpleType* oder *complexType* sein.

SchemaGlobalElement Schemaobjekte werden in XMLBeans noch weiter konkretisiert. Ein *SchemaGlobalElement* repräsentiert sogenannte *globale* Elemente, die in der Objekthierarchie unmittelbar auf ein *xs:schema*-Objekt folgen. Globale Elemente werden von *lokalen* Elementen unterschieden.

SchemaGlobalType Repräsentiert eine globale Typdefinition, die von Elementen in einem XML-Dokument über das Attribut *type* genutzt werden kann.

SchemaProperty Elemente und Attribute, die Elternelementen angehören, werden in *SchemaProperties* gruppiert und können zu jedem Element abgefragt werden. Es kann vorkommen, dass innerhalb einer komplexen Typdefinition verschiedene Elementtypen mit identischen Namen auftauchen. Solche Typen werden unter bestimmten Umständen zu einer einzigen Typdefinition zusammengeklappt, wenn die Unterschiede der Typdefinitionen am Ende keine Auswirkung auf das Endergebnis, dem XML-Dokumente, haben. In solchen Fällen *müssen* diese Typdefinitionen über *SchemaLocalElements* betrachtet werden ([9]).

SchemaLocalElement Repräsentiert lokale Elementedefinitionen, die in komplexe Typdefinitionen eingebettet werden. Der Informationsgehalt eines *SchemaLocalElements* kann über einen Durchlauf durch den sogenannten *SchemaParticle*-Baum abgerufen werden.

SchemaParticle Ein *SchemaParticle* repräsentiert eingebettete Objekte in Typdefinitionen. Solche Objekten können unter anderem *SchemaLocalElement*, *sequence*, *choice* und *all* sein. In einem *SchemaParticle* können andere *SchemaParticles* enthalten sein. Auf diese Weise wird ein komplexer Baum von *SchemaParticles* gebildet. Um zwischen den einzelnen Partikel-Objekten in einem Partikelbaum zu unterscheiden und ihre Informationen extrahieren zu können, muss ein *SchemaParticle*-Baum rekursiv durchlaufen werden.

Die *XsdIntrospection*-Klasse wird mit einem Array von Schemadefinitionen initialisiert. Aus diesem Array wird ein *SchemaTypeSystem* konstruiert. Über das *SchemaTypeSystem* werden alle Schemaobjekte abgerufen und einmal vollständig durchlaufen. Gefundene Informationen werden zu *QNames* gemappt, um einen unkomplizierten und zügigen Zugriff auf die Informationen zu einem *QName* zu ermöglichen.

In Listing 6.7 sind alle Felder der XsdIntrospection-Klasse aufgeführt, die für den generischen Editor von Bedeutung sind.

Listing 6.7: XsdIntrospection

```
1 public class XsdIntrospection {
2   private SchemaTypeSystem sts;
3   private HashMap<QName, SchemaGlobalElement> globalElements;
4   private HashMap<QName, SchemaType> globalTypes;
5   private HashMap<QName, SchemaProperty> attributes;
6   private HashMap<QName, SchemaProperty> childElements;
7   private HashMap<QName, Boolean> nillableElements;
```

```
8   private HashMap<QName, BigInteger> minOccurs;
9   private HashMap<QName, BigInteger> maxOccurs;
10  private HashMap<QName, Integer> positionInAllChoiceSequence;
11  private HashMap<QName, SchemaParticle> parentParticles
```

Die Hashtabellen *minOccurs*, *maxOccurs* und *nillableElements* speichern Informationen zu *SchemaLocalElements* ab, die nur über einen rekursiven Durchlauf durch den entsprechenden *SchemaParticle*-Baum erfasst werden können.

Über den Konstruktor (Listing 6.8) wird das *SchemaTypeSystem* auf Basis aller Schemadefinitionen erzeugt, die dem Editor im Ordner *schema* bereitgestellt werden. Die Überführung einer XSD-Datei in ein *XmlObject* ist trivial und an dieser Stelle nicht weiter ausgeführt.

Listing 6.8: XsdIntrospection: Konstruktor

```
1  public XsdIntrospection(XmlObject[] schemas) throws Exception {
2    sts = XmlBeans.compileXsd(schemas, XmlBeans.
         getBuiltinTypeSystem(), null);
3    /* initialization of hash maps omitted */
4    mapAllRelevantInfo();
5  }
```

Die Methode *mapAllRelevantInfo()* (Listing 6.9) leitet den Mappingprozess ein. Alle relevanten Informationen werden in diesem Schritt erfasst.

Listing 6.9: XsdIntrospection: mapAllRelevantInfo()

```
1  private void mapAllRelevantInfo() {
2    for (SchemaType st : sts.globalTypes()) {
3      globalTypes.put(st.getName(), st);
4      inspectElement(st);
5    }
6
7    for (SchemaGlobalElement globElem : sts.globalElements()) {
8      globalElements.put(globElem.getName(), globElem);
9      elementAnnotations.put(globElem.getName(),
           getAnnotationDocumentation(globElem.getAnnotation()));
10
11     // map child elements
12     for (SchemaProperty prop : globElem.getType().
           getElementProperties()) {
13       mapSubElement(prop);
14     }
15
16     // map attributes
17     for (SchemaProperty prop : globElem.getType().
           getAttributeProperties()) {
18       QName qn = prop.getName();
19       attributes.put(qn, prop);
```

```
20        }
21
22        inspectElement(globElem.getType());
23      }
24    }
```

Alle Schemaobjekte müssen dabei untersucht werden. Der Prozess beginnt bei globalen Typdefinitionen und globalen Elementen. Von dort aus angelt sich der Mappingprozess in der Hierarchie weiter nach unten. Für globale Elementedefinitionen werden *Annotationen* und *SchemaProperties* gemappt, welche die Definitionen von Kindelementen darstellen.

Elementedefinitionen können unter bestimmten Umständen zusammengeklappt werden. Deshalb wird für jeden globalen Typ und jedes globale Element der *SchemaParticle*-Baum durchlaufen (Listing 6.10), da nur so vollständige Informationen zu einer Typdefinition erfasst werden können.

Typdefinitionen treten in zweifacher Ausführung auf: als *globalType*, das über ein *type*-Attribut zu einem Element angegeben wird. Alternativ können Typdefinitionen in ein Element eingebettet werden.

Beide Fälle müssen berücksichtigt werden. Die Methode *inspectElement()* wird deshalb iterativ auf alle globalen Typdefinitionen und alle globalen Elemente angewandt (vgl. Listing 6.10).

Listing 6.10: inspectElement() und navigateParticle(); Quelle: [9]

```
1   private ArrayList<SchemaType> inspected = new ArrayList<>();
2
3   private void inspectElement(SchemaType t) {
4     if (null == t)
5       return;
6
7     if(inspected.contains(t))
8       return;
9     inspected.add(t);
10
11    if (t.getContentType() == SchemaType.ELEMENT_CONTENT ||
12            t.getContentType() == SchemaType.MIXED_CONTENT) {
13      SchemaParticle p = t.getContentModel();
14      navigateParticle(t, p);
15    }
16  }
17
18  public void navigateParticle(SchemaType t, SchemaParticle p) {
19    switch (p.getParticleType()) {
20      case SchemaParticle.ALL:
21      case SchemaParticle.CHOICE:
22      case SchemaParticle.SEQUENCE:
23        SchemaParticle[] children = p.getParticleChildren();
```

```
24          for (int i = 0; i < children.length; i++) {
25            SchemaParticle c = children[i];
26            parentParticles.put(c.getName(), p);
27            positionInAllChoiceSequence.put(c.getName(), i);
28            navigateParticle(c.getType(), c);
29          }
30          break;
31        case SchemaParticle.ELEMENT:
32          SchemaLocalElement localElement = (SchemaLocalElement) p;
33          QName qName = p.getName();
34          nillableElements.put(qName, p.isNillable());
35          minOccurs.put(qName, p.getMinOccurs());
36          maxOccurs.put(qName, p.getMaxOccurs());
37          SchemaAnnotation annotation = localElement.getAnnotation()
                ;
38          elementAnnotations.put(qName, getAnnotationDocumentation(
                annotation));
39
40          inspectElement(p.getType());
41          break;
42      }
43    }
44  }
```

Die Methode *navigateParticle()* arbeitet den Partikelbaum rekursiv ab und merkt sich dabei die jeweilige Partikelposition für *sequence*-, *all*- und *choice*-Objekte. Der W3C-Standard schreibt vor, dass die Reihenfolge, in der Kindelemente in einem Elternelement auftreten, der festgelegten Reihenfolge aus der Schemadefinition entsprechen muss ([1]).

Sobald *navigateParticle()* auf ein *SchemaLocalElement* trifft, werden *minOccurs*, *maxOccurs* und Annotationstext ausgelesen.

Anschließend wird der Prozess rekursiv über einen *inspectElement()*-Aufruf für dieses *SchemaLocalElement* wiederholt.

Alle besuchen Elemente werden vorgemerkt. Da sich zwei Typdefinitionen gegenseitig enthalten können, kann es beim *inspectElement()* zu einem rekursiven Überlauf kommen. Deshalb werden Elemente ignoriert, die bereits einmal besucht wurden.

7. Test

Für das Testen werden die Ergebnis-XML-Dokumente betrachtet, die aus der Speicherung einer Aufgabe resultieren.

Ein Vorteil der Anwendung ist, dass sie ihre Ausgaben selbst validiert. Durch eine automatisch ausgeführte XML-Validierung wird garantiert, dass Lehrkräfte XML-Dokumente erhalten, die *syntaktisch korrekt* und in Hinsicht auf verwendete XML-Schemadefinitionen *gültig* sind ([2]).

7.1. Testen der XML-Validierung

Um sicherzustellen, dass die XML-Validierung auch korrekt funktioniert, wurden Testfälle für das Testen der XML-Validierung erstellt. Hierzu werden erst einmal Negativtests eingesetzt, um sicherzugehen, dass die XML-Validierung nicht "versehentlich" funktioniert.

7.2. Testen inhaltlicher Werte

Damit sichergestellt wird, dass Eingabedaten einer Lehrkraft auf der Benutzeroberfläche korrekt in die XML-Dokumente übertragen werden, wurden Testfälle aufgestellt. Die Benutzerschnittstelle muss für das Editieren von ProFormA-Aufgaben getestet werden. Zum anderen muss geprüft werden, ob sich die Editiermechanismen für Elemente aus anderen Namensräumen korrekt verhalten.

Tabelle 7.1.: Negativtest der XML-Validierung für den ProFormA-Namensraum

Beschreibung des Testfalls	Eine Aufgabe so editieren, dass mindestens eine Musterlösung keine Dateireferenzen hat.
Erwartetes Ergebnis	Eine Fehlermeldung über unvollständigen Inhalt im *filerefs*-Element wird ausgegeben.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.2.: Negativtest der XML-Validierung für einen anderen Namensraum

Beschreibung des Testfalls	Die Funktionalität des Plugins aus Listing 6.3 wird so manipuliert, dass es fehlerhaft arbeitet und statt eines *Integers* einen *String* in das Feld *maxRuntimeSecondsWallclockTime* schreibt.
Erwartetes Ergebnis	Eine Fehlermeldung, dass der eingegebene Wert für den Datentyp *Integer* ungültig ist.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.3.: Positivtest der XML-Validierung

Beschreibung des Testfalls	Ein Positivtest soll die Validierung bei gültigen Eingabedaten für alle Namensräume erfolgreich durchlaufen lassen.
Erwartetes Ergebnis	Eine Ausgabe darüber, dass die Validierung erfolgreich durchgelaufen ist.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.4.: Validierung durch ein externes Tool

Beschreibung des Testfalls	Die Gültigkeit von erfolgreich validierten und gespeicherten XML-Dokumenten soll durch Zuhilfenahme eines externen Validierungs-Tools belegt werden. Hierzu wird IntelliJ für die Validierung verwendet.
Erwartetes Ergebnis	Eine Ausgabe darüber, dass die Validierung erfolgreich durchgelaufen ist.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.5.: Eingaben für Elemente aus dem ProFormA-Namensraum

Beschreibung des Testfalls	Alle Eingabefelder für den ProFormA-Namensraum ausfüllen.
Erwartetes Ergebnis	Alle Eingaben wurden in das XML-Dokument übertragen.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.6.: Eingaben durch Plugins

Beschreibung des Testfalls	Eingabedaten aus einem angebundenen Plugin sollen in das XML-Dokument übertragen werden. Getestet wird hierbei die Interaktion zwischen Host-Editor und Plugin, die bei einem Speichervorgang abläuft (vgl. Abbildung 5.6).
Erwartetes Ergebnis	Alle Eingaben wurden in das XML-Dokument übertragen.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.7.: Eingabedaten der generischen Editor-Komponente

Beschreibung des Testfalls	Vorgenommene Änderungen durch die generische Editor-Komponente sollen in das XML-Dokument übertragen werden. Getestet werden folgende Funktionen: • Das Hinzufügen von einem neuen Elementknoten • Das Hinzufügen von einem neuen Attributknoten • Das Löschen eines Elementknotens • Das Löschen eines Attributknotens • Die Änderung eines Knotenwertes
Erwartetes Ergebnis	Alle Änderungen an der Knotenstruktur und am Knotenwert wurden in das XML-Dokument übertragen.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

Tabelle 7.8.: Neue Elemente aus anderen Namensräumen hinzufügen und löschen

Beschreibung des Testfalls	Das Hinzufügen neuer Elemente aus anderen Namensräumen (vgl. Unterabschnitt 5.1.3) soll in das XML-Dokument übertragen werden. Analog dazu soll das Entfernen von existierenden Elementen aus dem XML-Dokument dazu führen, dass die Elemente auch tatsächlich entfernt werden.
Erwartetes Ergebnis	Für das Hinzufügen: neue Elemente wurden in das XML-Dokument an den richtigen Stellen angefügt. Für das Entfernen: betroffene Elemente wurden aus dem XML-Dokument an den richtigen Stellen entfernt.
Eingetretenes Ergebnis	Übereinstimmung mit erwartetem Ergebnis.

8. Zusammenfassung

Diese Arbeit hat das ProFormA-Format vorgestellt und einen Einblick in die Nutzung von ProFormA-Aufgaben gegeben. Aufgabenautoren stellen Aufgaben bereit, die von Lehrkräften angepasst und von Studenten anschließend bearbeitet werden. Lehrkräfte stellen die Nutzergruppe des Editors dar. Unter Berücksichtigung dieser Tatsache wurden entsprechende Anforderungen an den Editor ausgearbeitet, der schließlich konzipiert und implementiert wurde. Das Ergebnis ermöglicht die Bearbeitung von ProFormA-Aufgaben auf eine strukturierte und angenehme Weise. Die Bedeutung einzelner Elemente des ProFormA-Formats sind dokumentiert und können von der Lehrkraft als Hilfestellung abgerufen werden. Das Ziel, Erweiterungen aus anderen Namensräumen in ProFormA-Aufgaben editierbar zu machen, wird durch eine generische Editor-Komponente ermöglicht. Die benutzerfreundliche Darstellung von Erweiterungen kann über Plugins realisiert werden. Eine editorunterstützte Validierung von Eingabedaten sorgt dafür, dass Eingabefehler schnell und mühelos ausfindig gemacht und beseitigt werden können. Eine XML-Validierung garantiert, dass die resultierenden XML-Dokumente in Hinsicht auf verwendeten XML-Schemadefinitionen gültig sind.

Damit wurden alle Ziele erfüllt. Es ist ein neuer Editor entstanden, der sich auf die Bearbeitung von Programmieraufgaben im ProFormA-Format spezialisiert. Lehrkräfte können sich auf die Anpassung von Aufgaben konzentrieren, ohne von XML- oder ProFormA-Formaten abgelenkt zu werden.

Literatur

[1] Refsnes Data. *XML Schema sequence Element*. Abgerufen am 12.03.2017. URL: https://www.w3schools.com/xml/el_sequence.asp.

[2] Refsnes Data. *XML Validator*. Abgerufen am 02.04.2017. URL: https://www.w3schools.com/xml/xml_validator.asp.

[3] Peter Fricke, Robert Garmann, Felix Heine, Carsten Kleiner, Paul Reiser, Immanuel De Vere Peratoner, Sören Grzanna, Peter Wübbelt und Oliver J. Bott. „Grading mit Grappa - Ein Werkstattbericht.“ In: *ABP*. Hrsg. von Uta Priss und Michael Striewe. Bd. 1496. CEUR Workshop Proceedings. CEUR-WS.org, 2015. URL: http://dblp.uni-trier.de/db/conf/abp/abp2015.html#FrickeGHKRPGWB15.

[4] KBorm. *A Javascript editor for the exchange format for programming exercises*. Abgerufen am 23.02.2017. URL: https://github.com/ProFormA/formatEditor.

[5] ORACLE. *Java Architecture for XML Binding*. Abgerufen am 22.03.2017. URL: https://https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/.

[6] ORACLE. *javax.swing (Java Platform SE 8)*. Abgerufen am 11.03.2017. URL: https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html.

[7] o.V. *eCULT+*. Abgerufen am 17.03.2017. URL: www.ecult-niedersachsen.de.

[8] o.V. *XMLBeans 2.6.0 Documentation*. Abgerufen am 23.03.2017. URL: https://xmlbeans.apache.org/docs/2.6.0/reference/index.html.

[9] o.V. *XmlBeansFaq - Xmlbeans Wiki*. Abgerufen am 15.03.2017. URL: https://wiki.apache.org/xmlbeans/XmlBeansFaq.

[10] Sven Strickroth, Peter Fricke, Robert Garmann, Oliver Mueller, Oliver Rod und Uta Priss. *ProFormA specification and whitepaper*. Abgerufen am 23.02.2017. URL: https://github.com/ProFormA/taskxml.

[11] Sven Strickroth, Michael Striewe, Oliver Müller, Uta Priss, Sebastian Becker, Oliver Rod, Robert Garmann, Oliver J. Bott und Niels Pinkwart. „ProFormA: An XML-based exchange format for programming tasks“. In: *eleed* 11.1 (2015). Abgerufen am 23.02.2017. ISSN: 1860-7470. URL: `http://nbn-resolving.de/urn:nbn:de:0009-5-41389`.

A. CD