Felix Weise

Konzeption und prototypische Implementierung eines XML-Schema-gespeisten Java-Swing-GUI-Codegenerators zur Unterstützung einer effektiven Prototypenentwicklung

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Mathematik / Naturwissenschaften / Informatik

Mittweida, 2010

Felix Weise

Konzeption und prototypische Implementierung eines XML-Schema-gespeisten Java-Swing-GUI-Codegenerators zur Unterstützung einer effektiven Prototypenentwicklung

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Mathematik / Naturwissenschaften / Informatik

Mittweida 2010

Erstprüfer: Prof. Dr.-Ing. Wilfried Schubert
Zweitprüfer: Prof. Dr. rer. nat. Konrad Schulz

Vorgelegte Arbeit wurde verteidigt am:

Bibliografische Beschreibung:

Weise, Felix:
Konzeption und prototypische Implementierung eines XML-Schema-gespeisten Java-Swing-GUI-Codegenerators zur Unterstützung einer effektiven Prototypenentwicklung. – 2010. – 56 S. Mittweida, Hochschule Mittweida, Fachbereich Mathematik, Naturwissenschaften, Informatik. Diplomarbeit 2010.

Referat:

Ziel der Diplomarbeit ist es einen XML-Schema-gespeisten XML-Editor-Generator zu entwickeln. Der generierte XML-Editor basiert dabei auf einem Java-Swing-GUI. Ziel dieses Editors ist es, die OOA und Prototypenentwicklung dahingehend zu unterstützen, dass sie den Schritt vom Datenmodell zur grafischen Oberfläche erleichtert sowie die einfache Bearbeitung von XML-Dokumenten ermöglicht. Die Arbeit beschäftigt sich dabei mit den Grundlagen der Verarbeitung von XML und XSD, sowie den Verarbeitungsmethoden die dem Generator zugrunde liegenden. Anschließend wird die Funktion des Generators an einem Fallbeispiel schrittweise erläutert.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
DOM	Document Object Model
DTD	Document Type Definition
GUI	Graphical User Interface – Grafische Benutzeroberfläche
JAXP	Java API für XML-Verarbeitung
JSE	Abk. für Java Platform, Standard Edition
OOA	Objektorientierte Analyse
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSD	XML Schema Definition

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

1.1 Einführung

Die vorliegende Arbeit gewährt einen Einblick darin, wie man XML – Extensible Markup Language – und XML-Schema dazu nutzen kann, die Objektorientierte Analyse und Entwicklung zu unterstützen. Diese Unterstützung erfolgt in Form eines Code-Generators, welcher aus einem im Laufe der OOA entwickelten XML-Datenmodell einen individuellen, Java-Swing-basierten XML-Editor generiert.

Diese Arbeit entstand an der Hochschule Mittweida im Rahmen des Faches Softwaretechnik des Fachbereichs Informatik. Ihr Ziel ist es, einen Code-Generator zu entwickeln, welcher es ermöglicht, auf Basis von XML-Schema-Daten eine grafische Oberfläche zu generieren. Diese Oberfläche soll dann die Funktion eines einfachen XML-Editors erfüllen, welcher auf XML-Dateien, die auf dem gegebenen Schema basieren, zugeschnitten ist. Dieser kann dazu genutzt werden, die OOA zu unterstützen, indem er aus einem Datenmodell eine Eingabeoberfläche generiert und somit den Schritt von der Analyse zur prototypischen Implementierung erleichtert. Der generierte Editor kann sowohl als einfacher Editor ohne weiterführende Funktionen, als auch als Teil eines größeren Projektes eingesetzt werden.

Im Laufe der Arbeit wird sowohl ein Einblick in die Grundlagen der Bearbeitung eines XML- bzw. XSD-Dokumente gegeben, als auch mögliche Lösungen bestimmter Problemstellungen der Implementierung erläutert. Dabei konzentiert sich die Arbeit auf die Verarbeitung von XML-Daten in der Programmiersprache Java. Anhand des Code-Generators zeigt sie auf, was beim Umgang mit XML in Java beachtet werden muss.

1.2 Ziele der Arbeit

Die Ziele dieser Arbeit sind, die Möglichkeiten und theoretischen Grundlagen für auf XSD - XML Schema Definition - basierende Codegeneratoren aufzuzeigen, sowie einen entsprechenden Generator für einen XSD-basierten XML-Editor in Form eines Java-Swing-GUI-Prototypen zu entwickeln. Dieser Generator soll einen einfachen, Java-Swing basierten XML-Editor bereitstellen, welcher auf der Datenstruktur der XSD aufbaut. Die Syntax der verwendeten Schema-Dokumente entspricht dabei den Empfehlungen des W3C - das World Wide Web Consortium. Der generierte Editor soll dabei nur die Grundlagen bereitstellen, welche dann leicht erweiterbar sind, um sie den entsprechenden Anforderungen anzupassen. Es wird dabei keine Möglichkeit zur Bearbeitung der XSD bereitgestellt. Der Generator soll dabei keinen vollständigen grafischen Editor liefern, sondern lediglich einen Prototypen, welcher einfache Operationen zum Bearbeiten der jeweiligen XML-Datei bereitstellt. Alle weitergehenden Funktionen müssen vom Nutzer selbst angepasst werden. Dadurch soll ermöglicht werden, die Arbeit mit XML-Datenmodellen zu erleichtern. Dabei ist nicht Ziel, einen universellen XML-Editor-Generator zu erschaffen, sondern lediglich eine Hilfe, um auf bestimmte XML-Schemas zugeschnittene Eingabe-GUIs zu erstellen.

1.3 Aufbau der Arbeit

Diese Arbeit besteht im Wesentlichen aus den 3 Teilen: den theoretischen Grundlagen, der Erläuterung der Lösungsansätze auf theoretischer Basis und einem Fallbeispiel zur Verdeutlichung der Arbeit der Softwarelösung. Das 2. Kapitel beschäftigt sich mit den Möglichkeiten, die die XSD bietet, um als Grundlage für einen Codegenerator zu dienen. Es werden aber auch die Anforderungen, welche die XSD erfüllen muss, erläutert und damit die theoretische Grundlage der Lösungsansätze geschaffen. Im 3. Kapitel werden zuerst die theoretischen Lösungsansätze, die dem Generator zugrunde liegen, und die weiteren möglichen Lösungen beleuchtet, um die Funktionsweise des Generators und die Gründe für seinen Aufbau verdeutlichen zu können. Der Theorie folgt das Fallbeispiel einer Lagerverwaltung, welches die Vorgehensweise des Generators im Detail beschreibt und erklärt. Ebenfalls wird gezeigt, welche Schritte notwendig sind, um den GUI-Prototypen des Beispiels zu einer den Anforderungen entsprechenden Software zu erweitern und welche Anforderungen bereits durch den Generator erfüllt werden.

2 XSD als mögliche Präsentationsform für Datenstrukturen

2.1 Arbeiten mit XML-Schema

2.1.1 Was ist XSD?

XSD, die XML-Schema-Definition, ist eine Form von XML-Dokumenten zur Beschreibung und Validierung von anderen XML-Dokumenten. Im Vergleich zu den DTDs - den Dokumenttyp-Definitionen, welche nur grundlegende Strukturregeln vorgeben, sind XML-Schemas deutlich umfangreicher und anpasssungsfähiger. Ein mit einem bestimmten XML-Schema verbundenes XML-Dokument ist nur dann gültig, wenn es allen durch das Schema vorgegebenen Gültigkeitsregeln entspricht. Ob und welches XSD zur Beschreibung eines bestimmten XML-Dokuments verwendet wird, ist durch die XML-Datei selbst vorgegeben. Dies geschieht über das Attribut *xsi:schemaLocation* des Wurzelelements der XML. [Harold 05] S. 283 f.

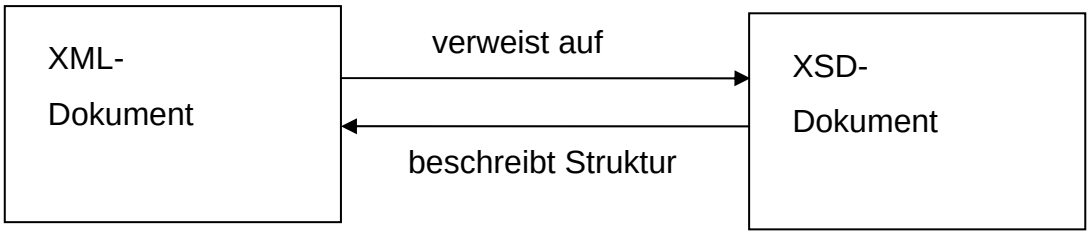


Abbildung 2.1 Verbindung XML und XSD

Dabei gilt zu beachten, dass jedes XSD-Dokument ebenfalls ein XML-Dokument ist und auch als solches gelesen werden kann.

2.1.2 Validierung von XML-Dokumenten die auf einer XSD basieren

Zur Validierung von XML-Dokumenten, welche auf einem Schema basieren, wird ein Parser benötigt. Dieser prüft, ob die XML-Datei wohlgeformt ist und allen Vorschriften ihres entsprechenden Schemas folgt.

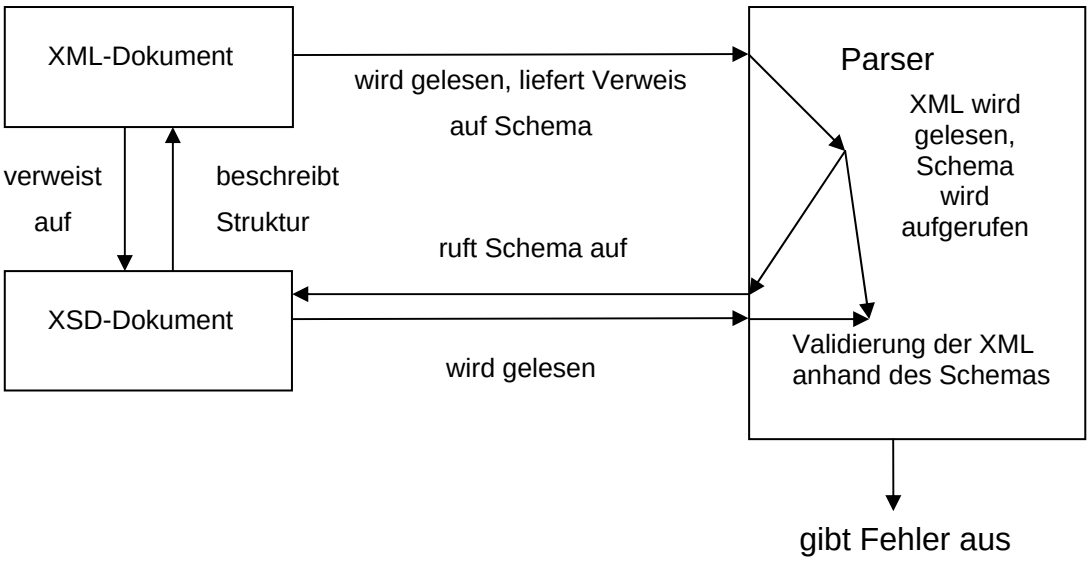


Abbildung 2.2 Validierung über Parser bei XML und XSD

Damit ein XML-Dokument validiert werden kann, wird eine fehlerfreie Schema-Datei vorausgesetzt. Im Dokument ist vor allem auf die Wohlgeformtheit und Gültigkeit zu achten. Dabei führen Verstöße gegen die Wohlgeformtheit dazu, dass das XML-Dokument nicht gelesen werden kann.

Die Regeln der Wohlgeformtheit beinhalten unter anderem:

- Jedes Element muss mit dem entsprechenden Endtag geschlossen werden
- Elemente dürfen sich nicht überlappen
- Es muss genau 1 Wurzelelement geben
- Attributwerte müssen immer in Anführungszeichen stehen
- Ein Element darf keine zwei Attribute mit demselben Namen haben

Nur wenn alle Regeln befolgt wurden, gilt das Dokument als wohlgeformt. Sollte der Parser im XML-Dokument auf Verstöße gegen die Regeln der Wohlgeformtheit stoßen, so bricht er die weitere Verarbeitung des Dokumentes ab. Dabei überprüft er den Rest des Dokumentes auf weitere Fehler und gibt diese zusammen mit den bisherigen aus. Das Beheben der aufgetretenen Fehler ist dabei die Aufgabe des Nutzers, der Parser selbst ist dazu nicht in der Lage. [Harold 05] S. 26, S. 30

Die Gültigkeit wiederum sagt aus, dass das XML-Dokument allen durch das Schema oder der DTD gegebenen Vorschriften folgt. Gültigkeitsfehler werden zwar durch den Parser erkannt, aber sie führen nicht zum Abbruch der Verarbeitung. Die entsprechenden Fehlermeldungen zu verarbeiten und die entsprechenden Fehler zu Beheben ist die Aufgabe der Software bzw. des Nutzers. [Harold 05] S. 30

2.1.3 Der Apache Xerces XML-Parser

Der Java-eigene Parser JAXP - Java API for XML Processing - war erstmals in JSE, der Java Plattform Standard Edition, Version 1.4 enthalten und ist in Version 1.6 durch den Apache Xerces Parser ersetzt. Ursprünglich aus einem 1999 an das xml.apache.org-Projekt verschenkten IBM-XML-Parser entstanden, ist der Apache Xerces der meist genutzte und am weitesten entwickelte Open-Source XML-Parser. Der Parser ist dabei sowohl für Java als auch für C++ und Perl, jeweils mit einer entsprechenden Version, verfügbar. Im Generator und in den entsprechend generierten Prototypen kommt er in der Version 2.9.1 zum Einsatz. Dabei ist wichtig zu beachten, dass Xerces Version 2.9.1 der Apache License Version 2.0 unterliegt. [Xerces]

2.1.4 Nutzbarmachen der XML- und XSD-Daten mithilfe des Parsers

Neben dem Validieren ist die Hauptaufgabe des Parsers, den XML-Inhalt in eine für Software nutzbare Form zu bringen: das DOM – Document Object Model. Dabei ist für den fehlerfreien Ablauf eine erfolgreiche Validierung der XML-Datei erforderlich. Um die Daten nutzen zu können, wandelt der Parser die einzelnen Elemente des XML-Dokuments in Objekte um, welche anschließend in einer Baumstruktur gespeichert werden.

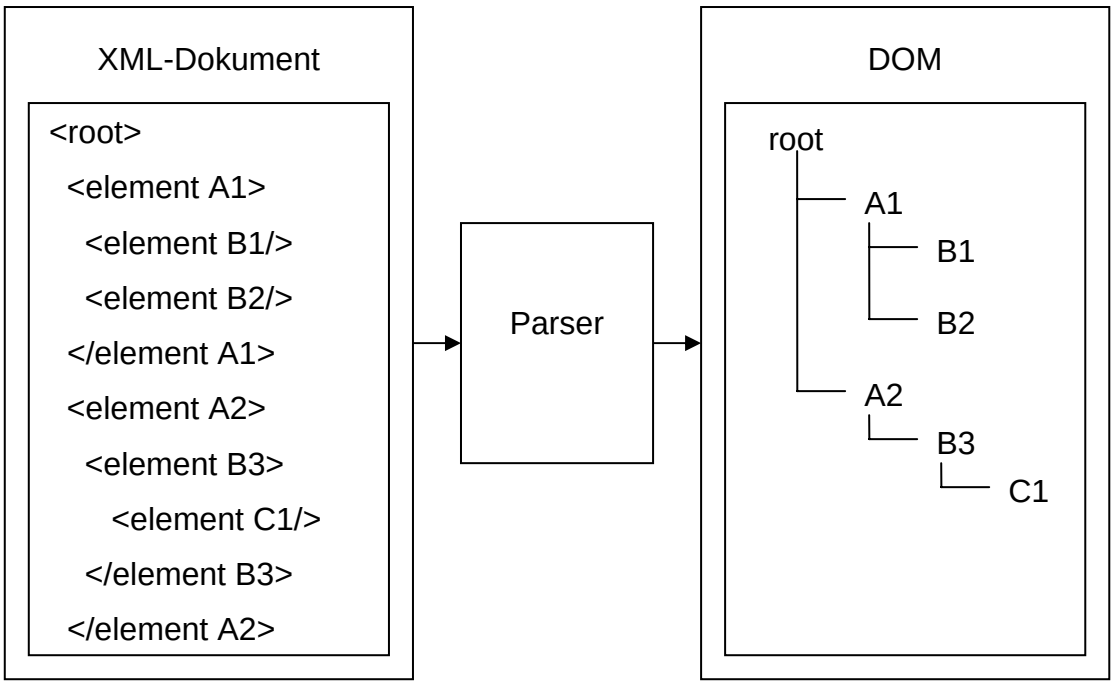


Abbildung 2.3 Umwandlung XML zu DOM

Die einzelnen Objekte innerhalb dieser Baumstruktur können nun von der Software genutzt und weiterverarbeitet werden. [Laughlin 06] S. 94 ff.

Beim Umwandeln eines XSD-Dokuments in ein DOM gibt es im Vergleich zum Umwandeln eines XML-Dokuments einige Unterschiede. Da die Schema-Dokumente vom Parser als XML-Dokumente eingelesen werden, findet die Validierung hier nur im Bezug auf die Wohlgeformtheit statt. Verstöße gegen die Gültigkeit werden vom Parser nicht gemeldet. Das führt dazu, dass bei der Fehlerkontrolle der Schema-Daten der Nutzer gefragt ist. XSD-Dokumente sind sehr verschachtelt aufgebaut. Dadurch wird die Baumstruktur des DOM ebenfalls entsprechend verschachtelt und tief, was bei der weiteren Bearbeitung beachtet werden muss.

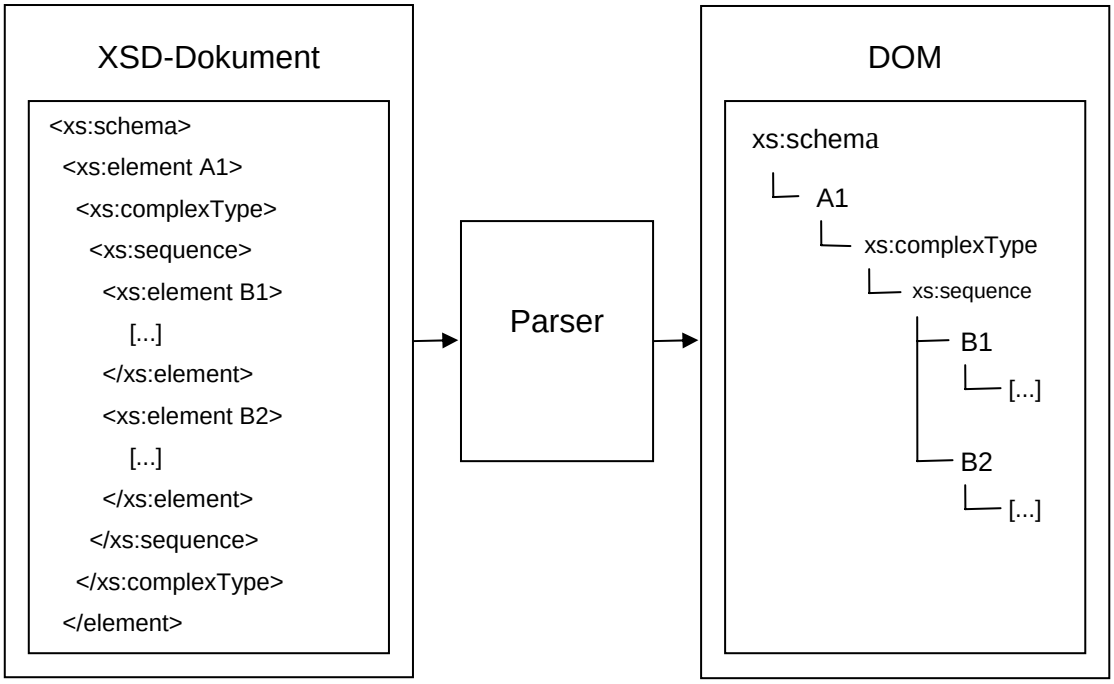


Abbildung 2.4 Umwandlung XSD zu DOM

Diese tiefe Verschachtelung führt zu einer aufwändigeren Verarbeitung als bei normalen XML-Dokumenten. Dabei muss man stets Abwägen, welche Elemente des DOM Daten enthalten und welche für die weitere Verarbeitung nicht benötigt werden.

2.2 Inhalte XSD, wie könnte man sie darstellen?

2.2.1 Welche Art von Daten kann man mit einer XSD beschreiben?

Wie bereits erwähnt ist die Hauptfunktion eines XML-Schemas, die Struktur einer XML-Datei zu beschreiben und festzulegen. Das reicht von der Anzahl und dem Namen der Elemente bis hin zu den Datentypen der einzelnen Bestandteile. Dabei werden die meisten gängigen Datentypen, aber auch die Möglichkeit zur Definition eigener Datentypen, bereitgestellt.

Die XSD selbst beschreibt dabei weniger die Daten, sondern eher die Darstellung der Daten. Durch die Definierung eigener Datentypen sowie dem frei wählbaren Aufbau der Schemas, innerhalb ihrer definierten Schema-Elemente, ist es möglich, dass diese Datendarstellung viele verschiedene Formen annimmt. Beispiele wären an erster Stelle Baumdiagramme, aber auch Tabellen und einfache Texte sind möglich.

2.2.2 Welche Art von Daten liefert die XSD selbst?

Ein Großteil der von einer XSD-Datei gelieferten Daten dient, wie bereits erwähnt, der Beschreibung einer XML-Datei. Diese Beschreibungen sind allerdings in der Art und Menge der Daten bzw. Strukturen, die sie beschreiben, eingeschränkt, da sie an die Definitionen ihrer entsprechenden Schema-Elemente gebunden sind.

Es ist allerdings möglich mithilfe des Elements *xs:annotation* Informationen zu hinterlassen, sowohl in Form einer Dokumentation *xs:documentation* als auch als Informationen für Software in form von *xs:appinfo*. Diese Informationen können, im Gegensatz zu einfachen Kommentaren, durch den Parser gelesen und auch an den Generator weitergegeben werden. Auf diese Weise könnte man zum Beispiel Elemente als „read only“ deklarieren, oder auch Masken, welche dann später auf die entsprechenden Werte angewendet werden, um beispielsweise IDs in eine bestimmte Form zu bringen. Auf diese Weise ist es möglich, zusätzliche Informationen zu einzelnen Elementen einzufügen, die in ihrer Art

und Struktur durch den Generator vorgegeben sind, anstatt durch die Schema-Definition. Dies hat natürlich den Nachteil, dass der Parser eventuelle Fehler innerhalb der Annotations nicht erkennt, bzw. diese vom Generator gesondert behandelt werden müssten. [Harold 05] S. 287 f.

2.2.3 Wie kann man die Daten einer XSD darstellen?

Da es sich bei einem XSD-Dokument um ein spezielles XML-Dokument handelt, ist es möglich die XSD wie ein XML-Dokument darzustellen. Der Aufbau einer XML-Datei macht eine Darstellung als Baum, in Form eines DOM, am naheliegendsten (vgl. Abbildung 2.3). Zwar kann man auch XSD-Dokumente als Baum darstellen, allerdings führt ihr deutlich verschachtelterer Aufbau dazu, dass diese Bäume viel tiefer sind und es entsprechend schwerer ist, darin zu navigieren und sie zu verarbeiten (siehe Abbildung 2.4).

Es ist daher sinnvoll, nach der Umwandlung des Dokuments in ein DOM, den Baum weiter zu bearbeiten, sofern das dem Ziel der entsprechenden Softwarelösung hilft. Der einfachste Schritt ist hierbei sicherlich das Kürzen des Baumes, so dass die für die Beschreibung eines XML-Dokuments wichtigen, für die Weiterverarbeitung aber hinderlichen Elemente entfernt werden.

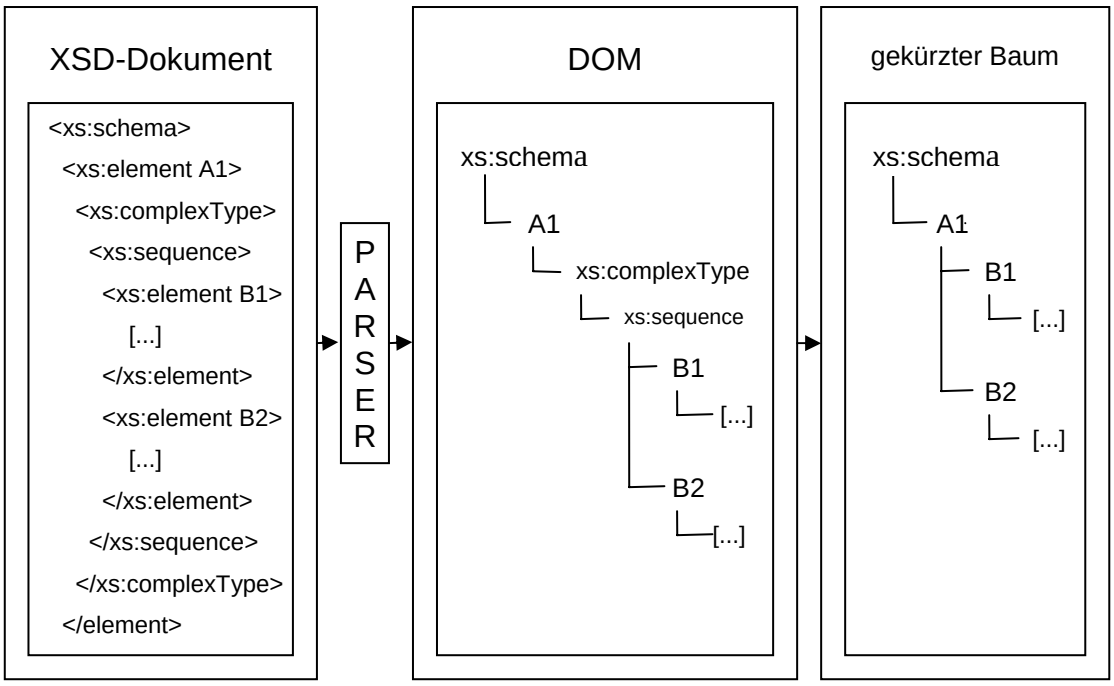


Abbildung 2.5 Umwandlung XSD zu DOM - Vereinfachung zum Baum

Wichtig ist dabei allerdings zu beachten, dass keine Daten verloren gehen. Außerdem führt die Vielzahl an möglichen Elementen und deren Bedeutung dazu, dass dieser Schritt, auch wenn er einfach aussieht, sehr aufwändig ist.

2.3 Datenkonsistenz, Validierung und Einschränkungen in XML, XSD und dem Generator

2.3.1 Datenkonsistenz

Die Aufgabe, die Datenkonsistenz eines XML-Schemas zu gewährleisten, liegt nahezu vollständig bei dem Benutzer, der das Schema liefert. Der Parser kann während der Validierung lediglich die Wohlgeformtheit des Dokuments überprüfen und auf falsche Datentypen hinweisen. Allerdings gibt es keinerlei Möglichkeit, den Inhalt des Schemas auf Richtigkeit zu überprüfen.

Analog zum XML-Schema liegt der Großteil der Sicherung der Datenkonsistenz beim Benutzer. Zwar kann der Parser, durch die Verbindung zu dem XML-Schema, falsch geformte Elemente oder falsche Inhalte, zum Beispiel Texte bei Integer-Werten oder ungültige Werte von Enumerationen, von Elementen aufzeigen. Allerdings kann er diese nicht korrigieren. Ebenso erkennt er keine inhaltlichen Fehler, die nicht auf die Datentypen zurückzuführen sind, zum Beispiel Fehler in Texten oder falsche Zahlenwerte.

In dem vom Generator gelieferten XML-Editor-Prototypen werden lediglich die vom Parser gelieferten Fehler wahrgenommen. Dabei spielen die Daten an sich für den Generator keine Rolle. Solange das Schema gültig ist und keine Formfehler aufweist, kann daraus ein Editor GUI generiert werden. Beim Lesen werden die Parserfehler gemeldet und, sofern nicht fatal, wird auf diese Fehler hingewiesen, damit der Nutzer sie korrigieren kann. Ebenso wird beim Schreiben neuer Daten auf die Konsistenz geachtet. Dabei wird auf auftretende Fehler hingewiesen und das Schreiben der Daten entsprechend abgebrochen. Die Fehler zu korrigieren und entsprechend richtige Daten einzugeben, liegt allerdings im Aufgabenbereich des Nutzers, der Editor kann hier keine automatischen Anpassungen durchführen.

2.3.2 Validierung

Die Validierung der XML- und XSD-Dokumente übernimmt der in Kapitel 2.1.3 vorgestellte Apache Xerces Parser. Dieser unterteilt gefundene Fehler in 3 Kategorien, *Warnings*, *Errors* und *FatalErrors*, wobei letztere zum sofortigen Abbruch führen.

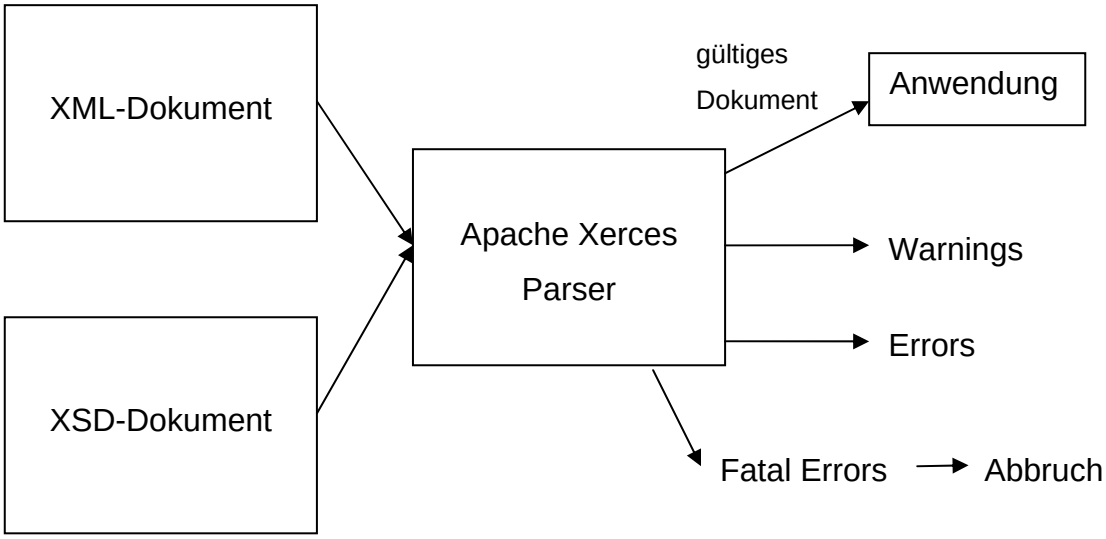


Abbildung 2.6 Validierung durch Parser

Die Art und Weise, wie der Parser mit Fehlern umgeht, kann man durch Einrichten eines individuellen *ErrorHandlers* verändern. Dadurch ist es möglich, die Ausgabeform der Fehler sowie den Umgang des Parsers mit den Fehlern zu beeinflussen. Auf diese Art ist es auch möglich, nicht validierbare Dokumente in der entsprechenden Software zu öffnen. [Xerces]

Zur Validierung einer XSD-Datei wird diese vom Parser als XML-Datei eingelesen und entsprechend auch nach den Regeln einer solchen Datei validiert. Einige geringere Fehler, hauptsächlich in den Schema-Definitionen, die durch die Unterschiede im Aufbau zwischen XSD- und XML-Dateien entstehen, müssen dabei entsprechend behandelt werden. Durch die Validierung des Schemas wird allerdings nur sichergestellt, dass diese der Form einer XML-Datei ohne Schema gerecht werden.

Damit der Generator einen Editor-Prototypen generieren kann, ist es wichtig, dass das Schema frei von allen Fehlern ist. Dieses wird zu diesem Zweck vali-

diert. Selbst eine Warnung führt hier zum Abbruch, da es zu unvorhersehbaren Ergebnissen kommen kann.

Der generierte Editor wiederum validiert die aktuell von ihm geladene XML-Datei bei jeder Output-Operation neu, um sicherzustellen, dass weder Fehler enthalten sind, noch neue Fehler eingebaut werden. Aber auch hier gilt: Falsche Eingaben müssen vom Nutzer korrigiert werden, der Editor selbst kann keine Daten korrigieren.

2.3.3 Einschränkungen

Die größte Einschränkung wurde in den vorangegangenen Kapiteln zu Datenkonsistenz und Validierung bereits mehrfach erwähnt. Weder der Parser, noch der Generator oder der Editor-Prototyp können Fehler von sich aus korrigieren. Egal ob es sich dabei um Formfehler oder inhaltliche Fehler handelt - die Verantwortung diese ordnungsgemäß zu beheben, liegt immer beim Nutzer. Auch muss man beachten, dass der Parser, und damit auch der Generator und der Editor, die XML-Dateien nicht auf falsche Daten überprüfen kann, außer diese haben falsche Datentypen. Hier liegt es wieder in der Verantwortung des Nutzers, dass dieser keine falschen Daten einträgt.

2.4 XSD und XML als Mittel der Datenspeicherung

2.4.1 XML als Datenformat für Datenbanken

Die Hauptaufgabe von XML-Dokumenten besteht meistens darin Daten zu speichern und zu beschreiben. Dabei bietet XML im Vergleich zu rationalen Datenbanken sowohl Vor- als auch Nachteile, so ermöglicht die Baumstruktur von XML-Datenbanken die einfache Serialisierung von hierarchischen Datenstrukturen. Im Gegensatz dazu sind relationale Datenbanken bei größeren Datenmengen meist effektiver. [Bourett]

2.4.2 XSD als Klassenmodell für XML-Datenbanken

Durch seine Funktion, das Beschreiben der Struktur und Art von Daten eines XML-Dokumentes, kann das Schema gleichzeitig als Klassenmodell für die Daten der XML-Datenbank angesehen werden. Es ermöglicht anderen Nutzern oder Softwarelösungen, einen Blick in die Struktur einer XML-Datenbank zu werfen bzw. deren Aufbau zu erkennen.

2.4.3 Die Verbindung zwischen Nutzer und XML-Datenbank

Um sicherzustellen, dass XML-Datenbanken nach der Bearbeitung durch Nutzer noch gültig sind, ist es nötig, die Datenbank erneut gegen das Schema zu validieren. An dieser Stelle setzt der Editor-Prototyp an, indem er eine Eingabeoberfläche bietet, welche sowohl die Bearbeitung des Inhalts der XML-Datenbank ermöglicht, als auch auf die Gültigkeit des Inhalts achtet.

Dadurch, dass der Generator den Editor direkt nach den Spezifikationen des Schemas erstellt, ist es möglich, durch den Editor selbst den Nutzer zu unterstützen und die Eingabe zu erleichtern.

Die Beziehung zwischen XML-Datenbank, Nutzer und GUI lassen sich folgendermaßen darstellen.

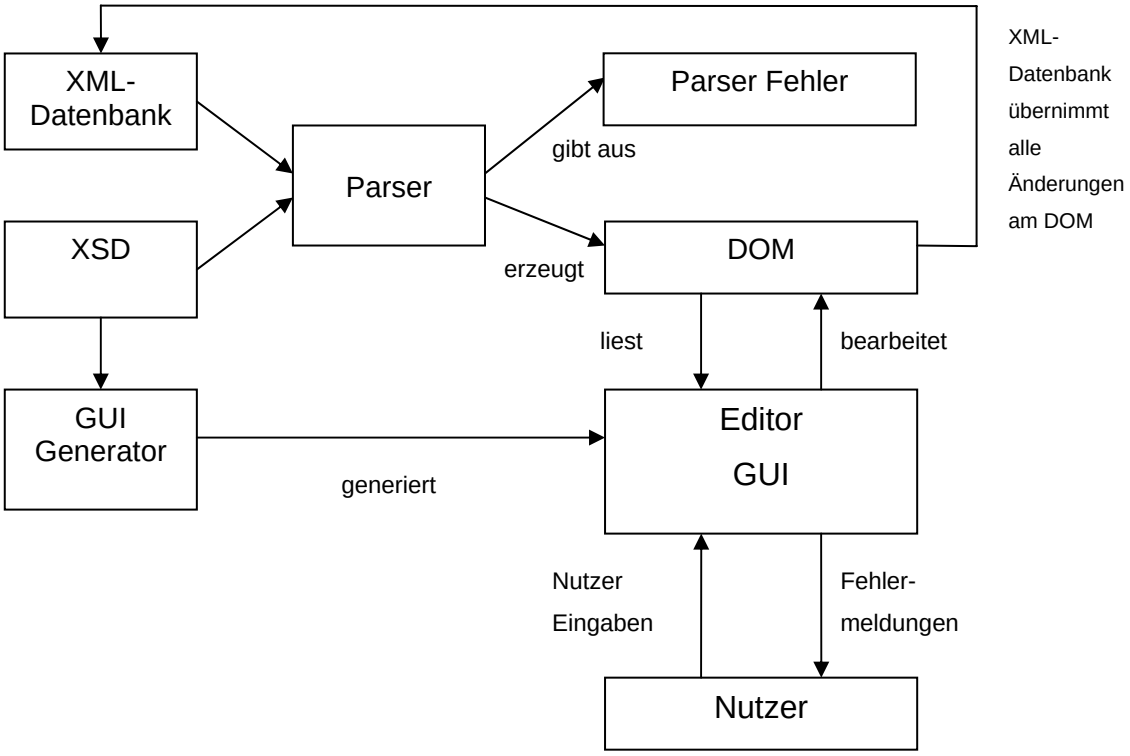


Abbildung 2.7 Schematische Darstellung XML-Datenbank - Nutzer - GUI

2.5 Beispiele bereits existierender XML-Editor Generatoren

Die Idee eines GUI-basierten XML-Editor -Generators ist nicht neu und es gibt dazu diverse kommerzielle und nicht-kommerzielle Softwarelösungen. Als Beispiel sollen an dieser Stelle zwei Generatoren genannt werden: *JAXFront* und *XAmple*.

JAXFront der xcentric technology & consulting GmbH bietet einen kommerziellen XSD-basierten GUI-Generator an.
„JAXFront is a technology to generate graphical user interfaces on multiple channels (Java Swing, HTML, PDF) on the basis of an XML schema. The dynamically generated GUIs allow the user a sophisticated way of editing XML data without being exposed to the underlying XML technology.“ [JAXFront]
Allerdings wird das GUI bei jedem Start der Applikation neu erstellt, und deren Spezifikationen können nur über ein entsprechendes Tool bearbeitet werden. Für die Ziele der Arbeit ist dieser Generator allerdings ungeeignet, da das von JAXFront generierte GUI nur bedingt an die vorliegenden Anforderungen angepasst werden kann.

Bei *XAmple* handelt es sich um einen XML-Schema gespeisten XML-Editor, welcher sowohl alleine lauffähig ist, als auch in ein entsprechendes Java-Projekt eingebunden werden kann.
„XAmple XML Editor project introduces a java Swing based XML editor that analyzes a given schema and then generates a document-specific graphical user interface.“ [XAmple]
Dabei gilt aber wie bereits bei JAXFront, dass sich der Editor selbst nicht bzw. nur wenig beeinflussen lässt, vor allem in Aussehen und Funktion.

Der Nachteil dieser Generatoren besteht meist darin, dass der Editor bei jeder Nutzung der Anwendung neu generiert wird. Das führt dazu, dass diese XML-Editoren meist in sich geschlossen und damit nur geringfügig anpassbar sind. Im Gegensatz dazu wird der durch den in dieser Arbeit entwickelten Generator erstellte XML-Editor beliebig anpass- und erweiterbar sein.

3 Entstehung eines GUI-Prototypen-Generators

3.1 Aufbau des Generators

3.1.1 Struktur des vom Generator-Prototypen erstellten Codes

Der Generator-Prototyp liefert einen Java-Swing-GUI XML-Editor welcher aus drei Klassen besteht: *Main*, *Control* und *View*.
Erstere wird vor allem die Initialisierungsfunktionen sowie die IO-Funktionen enthalten. Die *Control*-Klasse beinhaltet die *ActionListener* und wird die Kontrolle der verschiedenen GUI-Elemente übernehmen und die *View*-Klasse beinhaltet die GUI-Komponenten. Damit entspricht der generierte Editor dem *model*, *control*, *view* Konzept.

Nach erfolgreicher Codegenerierung befinden sich im vom Nutzer angegebenen Zielordner drei Java-Dateien, zwei Batch-Dateien sowie ein Verzeichnis. Bei den drei Java-Dateien handelt es sich um die drei Teile – *Main*, *Control* und *View* – des generierten Editors. Die zwei Batch-Dateien können dazu benutzt werden, um den generierten Editor-Code zu compilieren und zu starten. Das Verzeichnis *lib* enthält die für den Editor benötigten Bibliotheken, welche im Kapitel 3.1.3 kurz beschrieben werden.

3.1.2 Bestandteile des Generators

Der Generator selbst besteht aus sieben Klassen sowie einer *Main*-Klasse. Die Klasse *UIGenerator* ist dabei die Hauptklasse, welche den Grossteil der Generatorfunktionen enthält, der Code für jeden Teil des Editor-Prototypen wird jeweils von einer eigenen Klasse geliefert. Ebenso enthält der Generator mit der Klasse *GenElement* ein Datenmodell, auf welchem der Generator aufbaut. Dessen genaue Funktion und Aufbau wird im Kapitel 3.2.1 behandelt. Zum Generator gehören noch zwei weitere Klassen, *CopyFiles* und *CopyChanges*, welche Funktionen zum Kopieren von Dateien und zum Übernehmen von Änderungen in den Dateien zur Verfügung stellen. Der Genaue Aufbau der *UIGenerator* Klasse wird in Kapitel 3.3.1 beschrieben und ein vollständiges Klassendiagramm des Generators ist im Anhang zu finden.

3.1.3 Zusätzliche Bibliotheken

Der Generator bzw. der generierte Editor verwenden neben der Java 1.6 Bibliothek noch drei externe Bibliotheken: *xerces 2.9.1*, *jcalendar 1.3.3* und *SpringUtilities*.

Xerces ist der verwendete XML-Parser und wird dementsprechend immer benötigt. Er wurde bereits in Kapitel 2.1.3 genauer beschrieben.
JCalendar von Kai Tödter ist eine Freeware Java Bibliothek unter der GNU License welche es ermöglicht, einfach grafische Auswahlfenster und Eingabefelder für Datumseingaben einzubinden. Sie wird vom Editor verwendet, um Eingabefelder für Datumselemente zu realisieren. [JCalendar]
SpringUtilities ist eine ursprünglich aus dem Java-Tutorial stammende Klasse, welche das Konfigurieren des Spring-Layouts erleichtert. Sie wird vom Editor in der *View* genutzt und ist für die dynamische Größe und Füllung der Eingabepanels verantwortlich. [JavaTutorial]

3.2 Gegenüberstellung theoretischer Lösungsansätze und der Lösung des Generators

3.2.1 Wie der Inhalt einer XSD zum Erstellen eines GUI-Prototypen nutzbar gemacht werden kann

Zum Verdeutlichen der hier aufgeführten Methoden ziehen wir das Fallbeispiel einer kleinen Buchdatenbank heran.
Die entsprechende XSD-Datei sieht folgendermaßen aus:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="buecher">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="buch" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType mixed="true">
          <xs:sequence>
              <xs:element name="title" type="xs:string" />
              <xs:element name="author" type="xs:string" />
              <xs:element name="publisher" type="xs:string" />
              <xs:element name="ISBN" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Listing 3.1 Beispiel Schema - Bücher

Diese wird nun als eine ganz normale XML-Datei angesehen und dementsprechend vom Parser verarbeitet und nach dem DOM-Standard in ein Document Object umgewandelt. Dieses Objekt enthält den kompletten Inhalt der XSD-Datei in einer Baumstruktur, die im jetzigen Zustand bereits alles enthält, was der Generator benötigt, um den Prototypen zu erstellen. Allerdings enthält es auch viele für die Bearbeitung nicht benötigte Elemente und die Software müsste beim Einlesen immer wieder große Teile des im Objekt abgebildeten Baums durchlaufen. Deshalb ist es nötig, die Daten des Dokuments weiter zu bearbeiten und für das Erstellen des Prototypen zurechtzulegen. Es gibt dabei

verschiedene Herangehensweisen, die Vor- und Nachteile einiger davon werden im Folgenden erläutert.
Die drei hier beschriebenen Möglichkeiten sind: das Bestimmen der Position der benötigten Elemente innerhalb des DOM, das Speichern der benötigten Elemente außerhalb des DOM, Bearbeitung des DOM um schnelleren Zugriff zu ermöglichen.

Eine Möglichkeit wäre es, die für das Erstellen des Prototypen benötigten Teile des Dokuments zu ermitteln, um dem Generator zu ermöglichen, direkt auf sie zuzugreifen, statt jedes Mal den Baum durchlaufen zu müssen. Es spielt dabei keine Rolle, ob man die Position oder den Namen der Elemente benutzt, solange sich damit die Position des Elementes im Baum eindeutig bestimmen lässt. Der Vorteil hierin ist, dass der Generator das Objekt nur einmal kurz durchlaufen muss, um sich das wichtigste zu merken und sich dann den Rest während des Erstellens des Prototypen holt. Das führt dazu, dass weniger Speicher für tiefgreifende Analysen des Baums verbraucht wird. Allerdings hat dies auch Nachteile. Allen voran erhöht es die für das Erstellen des Prototypen benötigte Rechenzeit, da der Generator für jedes Element das Document Object heranziehen muss. Ebenso ist diese Methode anfällig für Fehler innerhalb der XSD. Da das Dokument nur kurz nach den wichtigsten Inhalten durchsucht wird, kann es passieren, dass zum Beispiel fehlende Typ-Attribute oder Angaben von Datentypen erst beim Erstellen des Prototypen erkannt werden und dort unweigerlich zum Abbruch führen.

Eine weitere Methode wäre es, statt wie in der vorher genannten Methode nur die Position oder den Namen, das gesamte entsprechende Element zu speichern und die dadurch entstandene Liste von benötigten Elementen an den Generator weiterzuleiten. Dadurch benötigt der Generator das Document Object nicht, sondern kann direkt mit der Liste der Elemente arbeiten und alle Daten daraus beziehen. Allerdings kann der Generator, wenn er ein bestimmtes Element benötigt, dieses nicht direkt ansprechen sondern muss dazu erst die Liste danach durchsuchen. Auch lassen sich nicht alle benötigten Informationen des XSD als Elemente abspeichern, was einige zusätzliche Variablen erfordert.

Es gibt auch die Möglichkeit, den im Document Object vorgegebenen Baum für den späteren Gebrauch zu stutzen und von unnötigen, beziehungsweise zum Erstellen des Prototypen nicht benötigten, Elementen zu befreien. Beispiele hierfür wären *<complexType>* oder *<simpleType>*, welche zwar in der XSD benötigt werden, aber für den Generator eher unwichtig sind, da er die entsprechenden Informationen aus der Struktur des Baumes erkennt. Bei dieser Methode wird der Baum durchlaufen und jedes nicht benötigte Element entfernt. Das ermöglicht dem Generator später beim Erstellen des Prototypen, den Baum schneller zu durchlaufen, und er muss auch nicht mehr überprüfen, ob er das jeweilige Element benötigt oder es ignorieren kann. Problematisch ist nur, dass auch hier einige Informationen gesondert gespeichert werden müssen.

Die Vor- und Nachteile der einzelnen Methoden werden im Folgenden zusammengefasst.

Vorteile	Nachteile
Position der Elemente im Dom ermitteln:	
- keine Mehrfache Analyse des DOM notwendig - Schneller Zugriff auf die Inhalte im DOM	- Anfällig für Fehler in der XSD-Datei - Erhöhter Rechenaufwand durch häufige Zugriffe auf das DOM
Benötigte Elemente außerhalb des DOM speichern:	
- keine Mehrfache Analyse des DOM notwendig - DOM wird nach Analyse nicht mehr benötigt	- Elemente können nicht direkt aufgerufen werden - Erhöhter Rechenaufwand durch die Suche nach bestimmten Elementen - Zusätzliche Variablen nötig
Bearbeiten des DOM für schnelleren Zugriff:	
- Zeit für die Analyse des DOMs ist deutlich geringer - Elementspezifikationen können schon während der Analyse gespeichert werden	- Zusätzliche Variablen nötig - Erhöhter Rechenaufwand durch häufige Zugriffe auf das DOM

Im Generator selbst wird eine Mischung aus der zweiten und der zuletzt genannten Methode verwendet. Das bedeutet, die Elemente des DOM werden in einer neuen Datenstruktur in Baumform gespeichert. Dabei werden die Informationen der nicht benötigten Elemente an die entsprechenden Eltern- oder Kindelemente. Hierzu wurde ein eigenes Objekt erstellt, welches als Grundlage für die Baumstruktur genutzt wird und die wichtigsten Informationen direkt bereitstellt.

Der Prototyp dieses Objekts *GenElement* hat die folgende Form

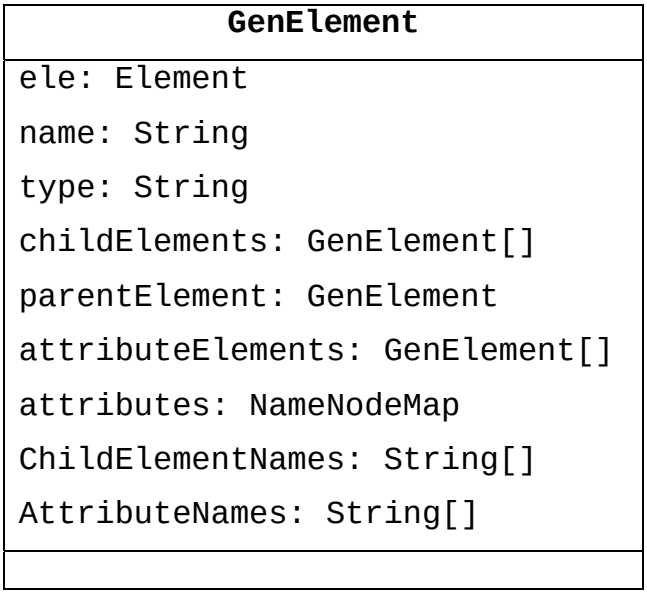


Abbildung 3.1 Objekt GenElement zur Weiterverarbeitung von DOMs

Das Element *ele* enthält das ursprüngliche Element, wie es im Document Object vorliegt.
Der String *name* enthält den Namen des Elements, im Bezug auf das Fallbeispiel zum Beispiel *buecher* oder *buch*. Dies ist eine der wichtigsten Informationen für den Generator.
Der String *type* wird genutzt um anzugeben, ob es sich bei dem jeweiligen Element um die Wurzel, einen Ast oder ein Blatt des Baumes handelt. Hieran erkennt der Generator die jeweilige Bedeutung des Elements.
ChildElements enthält alle Kindelemente des aktuellen Elements. Im Fallbeispiel enthält das *GenElement buch* hier die vier *GenElemente title*, *author*, *publisher* und *ISBN*.
ParentElement wird genutzt, um ein Zurückgehen im Baum zu ermöglichen.
AttributeElements enthält alle Kindelemente, die Attribute beschreiben.

Die NamedNodeMap *attributes* enthält alle in der XSD vorhanden Attribute des jeweiligen Elements. Für *buch* wären dies *name*, *maxOccurs* und *minOccurs*.
Die beiden String-Arrays *ChildElementNames* und *AttributeNames* stellen den Namen der jeweiligen Kindelemente zur Verfügung. Dies verringert den Rechenaufwand bei Generatorfunktionen, die lediglich den Namen der Elemente benötigen und diese so nicht direkt aufrufen müssen.
Mithilfe dieses Objektes lässt sich der Inhalt der XSD in eine einfache Baumform bringen, ohne dass Informationen verloren gehen oder nicht mehr zugänglich sind. Zur Weiterverarbeitung im Generator wird diesem das Wurzelelement übergeben und dieser greift darüber direkt auf die benötigten Daten und Kindelemente zu. Dieses Objekt wurde im weiteren Verlauf der Entwicklung erweitert, der hier gezeigte Prototyp enthält aber bereits alle wesentlichen Elemente.

Um die Daten aus der XSD in das Objekt zu übertragen wird nun der Baum des Document Object durchlaufen und Element für Element auf seinen Informationsgehalt und seine Bedeutung bei der Erstellung des GUI-Prototypen untersucht. Die entsprechenden Daten werden dabei in den neuen Baum aus *GenElement*-Objekten übertragen.
Im Falle der einfachen Buchdatenbank würde der Baum aus den folgenden *GenElement*-Objekten und den entsprechenden Beziehungen bestehen.

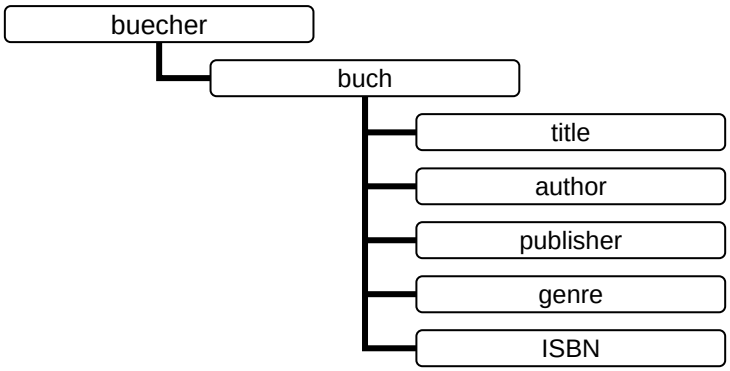


Abbildung 3.2 Baumdiagramm - Bücher

3.2.2 Möglichkeiten der Codegenerierung auf Basis der in der XSD enthaltenen Daten

Die Daten der XSD wurden vom Generator gelesen und in eine entsprechende Baumstruktur übertragen. Nun muss aus diesen Daten Code generiert werden. Hierbei sollte auch beachtet werden, dass neben den von den XSD-Daten abhängigen Codefragmenten auch von den Daten unabhängige Teile generiert werden müssen.

Die einfachste Möglichkeit besteht darin, den Code Zeile für Zeile in die entsprechende Zieldatei zu schreiben. Dabei werden an den von der XSD abhängigen Codezeilen zuerst die entsprechenden Daten aus dem Baum gelesen und mit diesen die jeweilige Codezeile gebaut. Das Problem an dieser Methode besteht darin, dass der für das Schreiben des Codes zuständige Teil des Generators nur schwer editier- und erweiterbar ist. Dies würde spätere Anpassungen des Generators erschweren.

Eine wesentlich weniger aufwändige Möglichkeit wäre es, den Code in Abschnitte zu unterteilen, welche dann vom Generator je nach Bedarf abgerufen werden. Dabei werden, in den entsprechenden von XSD-Daten abhängigen Abschnitten, die zusätzlichen Daten in die bereits vollständigen Codefragmente eingefügt. Im Grunde entspricht dies der bereits erwähnten Zeile-für-Zeile-Methode. Allerdings benötigt man dadurch, dass keine Codezeilen sondern ganze Codeblöcke auf einmal geschrieben werden, weniger Ausgabe-Operationen und kann so Ausgabefehler leichter lokalisieren. Auch lassen sich diese Code-Abschnitte leichter bearbeiten oder erweitern.
Diese Methode der Codeblöcke, als eine Form der Codegenerierung, könnte man sogar dahingehend erweitern, dass man den entsprechenden Code nicht in der Software selbst, sondern in einer externen XML-Datei speichert. Diese XML-Datei wird im Generator eingelesen, welcher sich dann den entsprechenden Code nicht mehr als String selbst erstellt, sondern direkt aus der XML- in die Zieldatei überträgt. Zusätzlich würde dies dem jeweiligen Nutzer ermöglichen, Codefragmente oder Bezeichnungen dauerhaft, statt nur im jeweiligen Projekt, nach seinen Wünschen anzupassen.

Vorteile	Nachteile
Code Zeile für Zeile schreiben:	
- Einfache Funktionsweise - Geringer Rechenaufwand da immer nur kurze Zeilen direkt geschrieben werden	- Hoher Aufwand in der Programmierung - Nur schwer Erweiter- oder Editierbar
Code in Form von Blöcken schreiben:	
- Einfach Erweiter- oder Editierbar - Leicht auf andere Sprachen zu übertragen - Weniger Ausgabeoperationen nötig	- Hoher Speicheraufwand während des Generierens

Der Generator selbst besitzt in seinem Code alle benötigten Codefragmente, die je nach Bedarf zu mehreren Strings zusammengefügt werden, welche miteinander verbunden den Quellcode bilden. Der Quellcode und damit auch der generierte Editor ist dabei, wie in Kapitel 3.1.1 erwähnt, auf drei Klasseb verteilt: *Main*, *Control* und *View*. Wenn der Code einer Klasse erfolgreich generiert wurde, wird er in eine temporäre Datei geschrieben, welche später in den Zielordner kopiert wird.

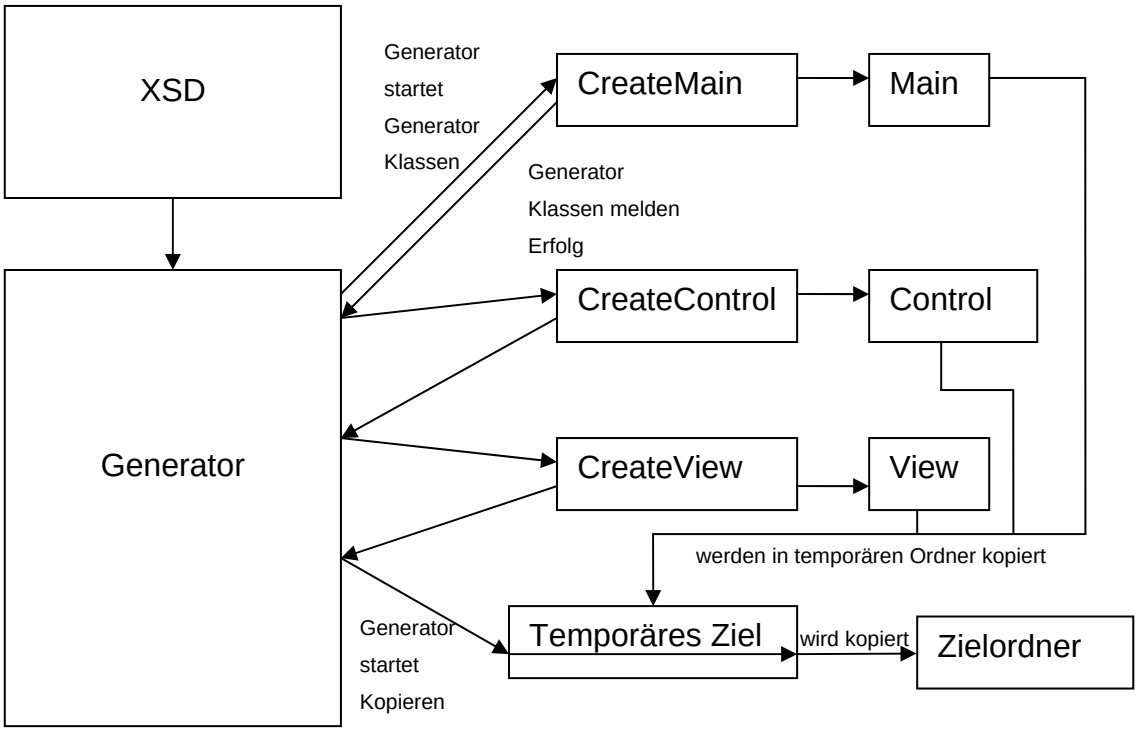


Abbildung 3.3 Generierung Allgemein

Die meisten Grundfunktionen des Editors, z.B. das Laden einer XML-Datei oder der Parser, sind unabhängig von den Elementen der XSD und sind deshalb bei jedem Generier-Prozess gleich. Funktionen, welche abhängig von den XSD Elementen sind, werden zusammengesetzt, indem rekursive Funktionen die Baumstruktur durchlaufen und für jedes entsprechende Element einen bestimmten Code liefern.

Am deutlichsten wird dies in der *View*, welche für verschiedene Datentypen andere GUI-Komponenten liefert. Als Beispiel sollen hier die vier verschiedenen Code-liefernden Funktionen für s*tring*-, *integer*-, *double*- und *boolean*-Elemente dienen. Sie definieren jeweils ein *JLabel*, welches im späteren GUI vor der jeweiligen Komponente steht, sowie das zum Datentyp passende Eingabefeld. *JTextField* für *string*, *JTextField* mit spezieller Formatierung für *integer* und *double* sowie eine *JCheckBox* für *boolean*.

```
private String get_View_definition_textfield(){
  return "\tpublic JLabel <<element_name>>_<<node_name>>_lbl =
                                          new JLabel(\"<<node_name>>\");\n"
       + "\tpublic JTextField <<element_name>>_<<node_name>>_txt;\n";
}
private String get_View_definition_textfield_integer(){
  return "\tpublic JLabel <<element_name>>_<<node_name>>_lbl =
                                          new JLabel(\"<<node_name>>\");\n"
       + "\tpublic NumberFormat <<element_name>>_<<node_name>>_format =
                                       NumberFormat.getNumberInstance();\n"
       + "\tpublic JFormattedTextField <<element_name>>_<<node_name>>_txt;\n";
}
private String get_View_definition_textfield_double(){
  return "\tpublic JLabel <<element_name>>_<<node_name>>_lbl =
                                          new JLabel(\"<<node_name>>\");\n"
     + "\tpublic NumberFormat <<element_name>>_<<node_name>>_format =
                                       NumberFormat.getNumberInstance();\n"
     + "\tpublic JFormattedTextField <<element_name>>_<<node_name>>_txt;\n";
}
private String get_View_definition_checkbox(){
  return "\tpublic JLabel <<element_name>>_<<node_name>>_lbl =
                                          new JLabel(\"<<node_name>>\");\n"
       + "\tpublic JCheckBox <<element_name>>_<<node_name>>_ckb;\n";
}
```

Listing 3.2 Generator Code createView Ausschnitt

Nach der Rückgabe der Codefragmente an die entsprechende Funktion werden die zusätzlichen Informationen, welche durch die jeweiligen Elemente bestimmt werden, an den entsprechenden Stellen eingefügt. Diese Stellen sind jeweils mit <<*…*>> markiert, z.B. steht <<*element_name*>> für den Namen des Elements, dessen Kindelemente die einfachen Datentypen bilden. Im Falle der einfachen Buchdatenbank wäre dies das Element *buch*, während <<*node_name*>> *title*, *author*, *publisher* etc. entsprechen würde.

3.2.3 Neuen Code in bereits vorhandene Dateien einfügen

Um den Codegenerator effektiv in den Entwicklungsprozess einer Software einbeziehen zu können, ist es nötig eine Funktion bereitzustellen, die es ermöglicht Änderungen an dem XML-Schema umzusetzen, ohne dabei bereits bearbeiteten Code zu überschreiben. Die Möglichkeiten der Umsetzung dieser Funktion die im Folgenden beschrieben werden sind: das Abgrenzen des generierten Codes innerhalb der Dateien, der Vergleich des generierten Codes mit dem unbearbeitetem vorherigen Code, der Vergleich des generierten Codes mit dem bearbeiteten Code.

Die einfachste Möglichkeit wäre innerhalb der generierten GUI-Dateien bestimmte Bereiche zu markieren, beispielsweise durch Kommentare, welche anschließend beim Einfügen von neuen Daten erhalten bleiben. Dies wäre allerdings nur dann praktikabel, wenn die Grundfunktionen und Elemente des GUI komplett unverändert genutzt werden. Hinzu kommt, dass diese Lösung für den späteren Nutzer eher unpraktisch ist, da man beim Bearbeiten des GUI Codes sehr vorsichtig vorgehen und darauf achten müsste, die Generator-Kommentare nicht zu bearbeiten.

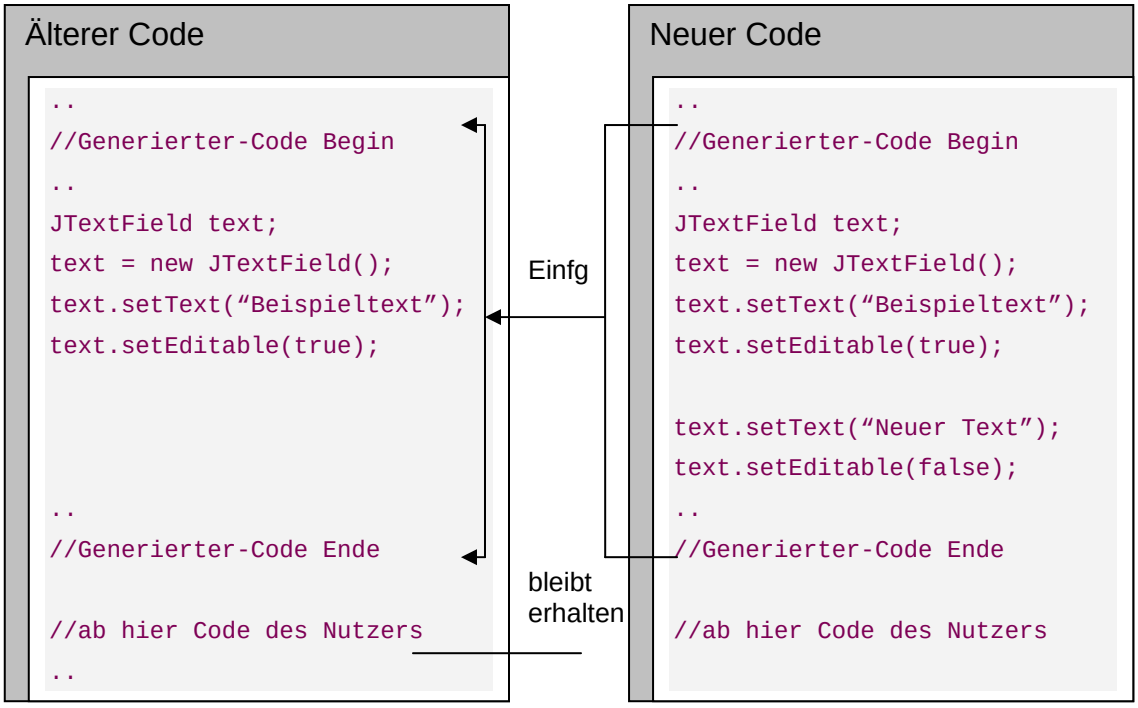


Abbildung 3.4 Code Einfügen Methode 1 - Kommentare als Begrenzer

Eine deutlich aufwändigere aber dabei auch leistungsfähigere Möglichkeit ist das Vergleichen des bereits vorhandenen Codes mit dem neu generierten. Allerdings wirft dies weitere Probleme auf, z.B. wie der Generator mit bereits bearbeitetem Code umgehen soll. Dieses Problem könnte durch das Einbinden einer „Generator-Historie“ umgangen werden. Dabei würde jedes generierte GUI vom Generator gespeichert und bei jeder neuen Generierung würde der neue Code mit dieser vorherigen verglichen, um die Änderungen zu ermitteln. Diese Methode ist allerdings sehr aufwändig und die „Historie“ wirft neue Probleme auf. Der Generator lässt hier die Änderungen im vorhanden Code außer Acht, was im Grunde dazu führt, dass letztendlich wieder mit dem bereits editierten Code verglichen werden muss, um die genaue Position der Änderungen zu ermitteln.

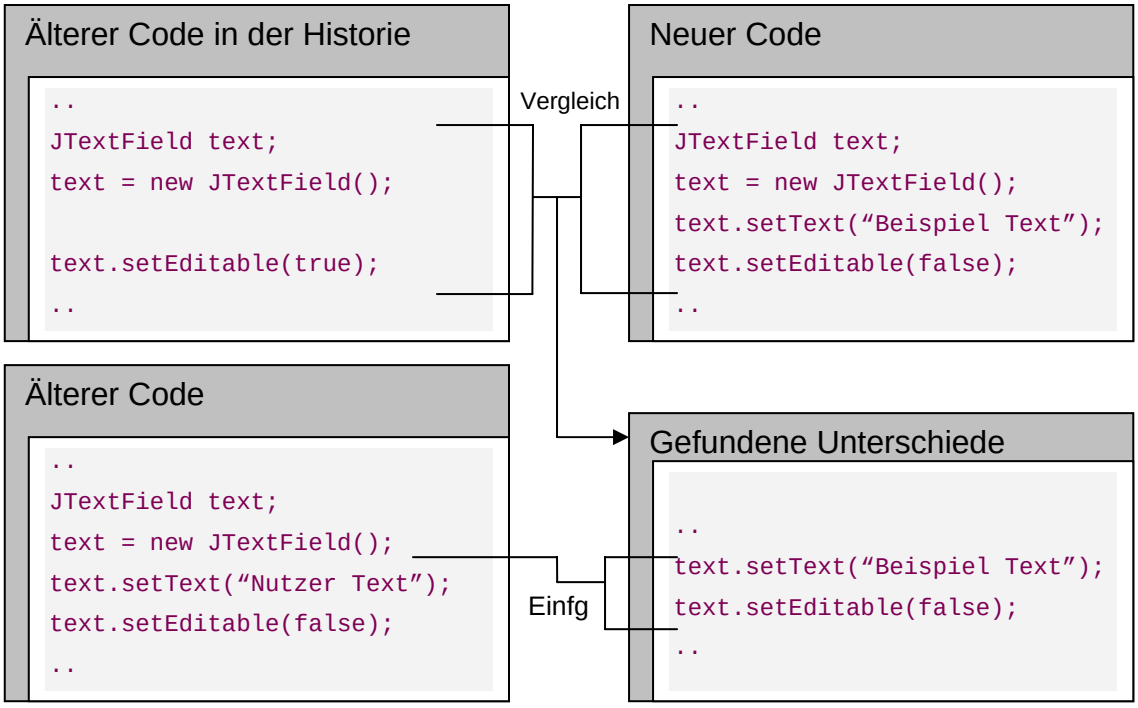


Abbildung 3.5 Code Einfügen Methode 2 - Vergleich mit gespeicherter Historie

Die dritte Methode, welche der Generator selbst verwendet, vergleicht den generierten Code mit dem bereits vorhandenen Zeile für Zeile. Sobald eine Zeile in der neu generierten Datei nicht der Zeile in der vorhandenen entspricht, wird die vorhandene Datei nach dieser Zeile durchsucht. Sollte sie gefunden werden, wird das Vergleichen an dieser Stelle fortgesetzt. Wenn eine Zeile der neuen Datei in der existierenden nicht vorhanden ist, wird diese entsprechend eingefügt; dabei ist die Positionierung der aufwändigste Teil. Um sicherzugehen, dass die Position des neuen Codes auch mit seiner vorhergesehenen übereinstimmt, vergleicht der Generator die der neuen Zeile folgenden Zeilen auf Übereinstimmung mit dem vorhandenen Code. Dabei ist es allerdings wichtig, dass häufig auftretende Zeilen entsprechend behandelt werden. Dazu werden diese einfach mit den weiter nachfolgenden Zeilen verknüpft. Diese Methode hat allerdings auch Nachteile, welche vom Nutzer entsprechend beachtet werden müssen. Wenn generierter Code geringfügig geändert wurde, z.B. durch Umbenennung einer Variablen, führt dies beim erneuten Ausführen des Generators dazu, dass die alte Bezeichnung in unmittelbarer Umgebung der geänderten eingefügt wird.

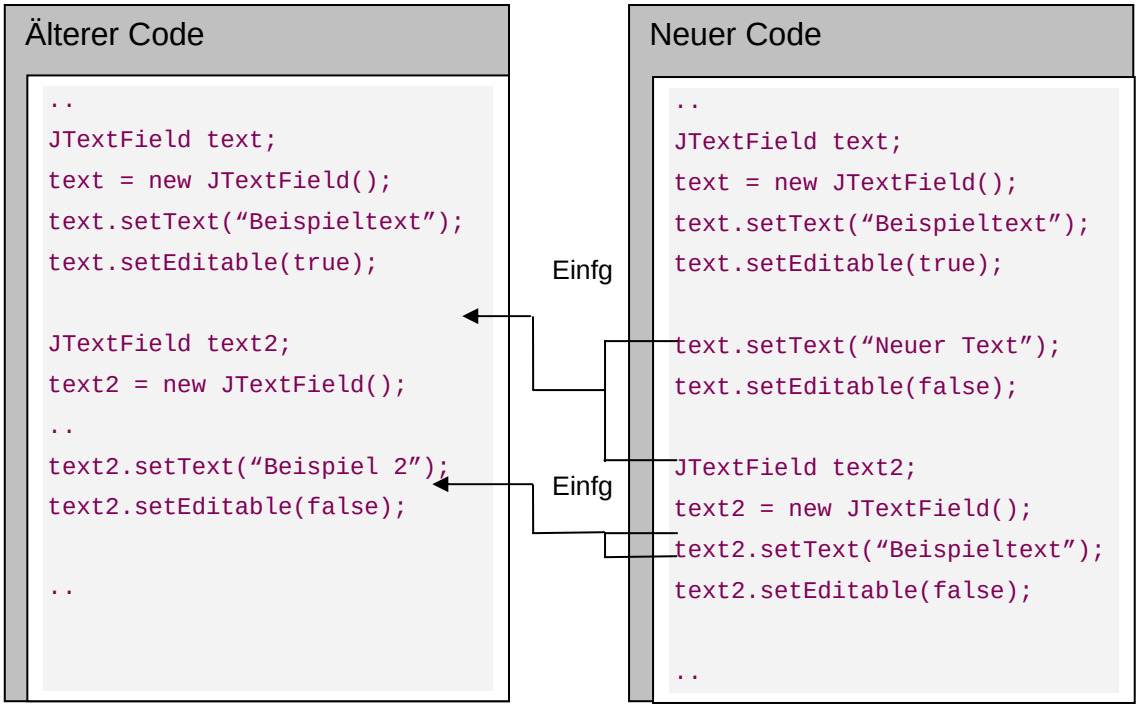


Abbildung 3.6 Code Einfügen Methode 3 - Vergleich mit bearbeitetem Code

Auch können sehr umfangreiche Änderungen am Code oder dessen Formatierung zu unvorhergesehenen Ergebnissen führen, weshalb diese Funktion im fortgeschrittenen Stadium der OOA nur bedingt einsetzbar ist.

Wägt man die Vor- und Nachteile der einzelnen, beschriebenen Methoden ab, wird allerdings klar, dass keine der drei hier beschriebenen Methoden ideal ist. Dabei bietet die 3. Methode im vergleich zu den andern die größere Funktionalität und ist durch den Nutzer am einfachsten zu verwenden, weshalb sie im Generator zum Einsatz kommt.

Vorteile	Nachteile
Überschreiben bestimmter mit Kommentaren Markierter Bereiche:	
- Schnelles Einsetzen des neuen Codes	- Die Kommentare sind notwendig damit dies funktioniert - Änderungen direkt im generierten Code sind nicht möglich
Vergleich mit „Historie“ und übernehmen der Unterschiede:	
- Bearbeiteter Code wird nicht verändert	- Angelegte Historie muss gepflegt werden - Probleme wenn der bearbeitete Code zu sehr von Historie abweicht - Keine Möglichkeit gelöschten Code wiederherzustellen
Vergleich mit der bearbeiteten Datei und übernehmen der Unterschiede:	
- Bearbeiteter Code wird nicht verändert - Wiederherstellung gelöschten Codes möglich	- Kleinere Änderungen am Code führen beim erneuten generieren zum einfügen der alten Zeile vor der bearbeiteten - Vom Nutzer gelöschter Code wird immer neu eingefügt - Relativ aufwendiger Vergleichs- und Einfüge-Prozess

3.2.4 Validierung und Fehlerbehebung

Bei der Generierung des GUI-Prototypen spielt die Validierung eine untergeordnete Rolle, da die XSD-Dokumente selbst keine Schemas besitzen, nach denen sie validiert werden könnten. Sie beschränkt sich deshalb beim Generator auf jene Fehler, welche eine Verarbeitung der XSD unmöglich machen: Verstöße gegen die Wohlgeformtheit. Da diese aber nicht durch den Generator behoben werden können, ist hier die fehlerfreie Vorlage des Nutzers nötig.
Im Gegensatz zum Generator ist im generierten XML-Editor die Validierung der eingegebenen Daten von großer Bedeutung. Bei der Arbeit an XML-Dokumenten ist es vor allem wichtig, darauf zu achten, dass diese auch mit den vorgenommenen Änderungen noch gültig sind. Ein falscher Datentyp oder ein falsches Element können dazu führen, dass das Dokument nicht mehr validiert und damit durch die Software nicht mehr gelesen werden kann. Das generierte GUI ist in seiner unbearbeiteten Form dabei sehr großzügig bei der Validierung, so dass nur die schwersten Fehler zum Abbruch führen. Dennoch muss man sichergehen, dass die gerade mit dem GUI vorgenommenen Änderungen an einem XML-Dokument dessen Gültigkeit nicht beeinträchtigen. Zu diesem Zweck wird das DOM herangezogen. Alle Änderungen werden zuerst am DOM vorgenommen, welches später wieder in das XML-Dokument umgewandelt wird. Um seine Validierbarkeit sicherzustellen wird eine temporäre Kopie des DOM erstellt, welche wiederum vom Parser verarbeitet wird. Jegliche auftretenden Fehler werden zurückgegeben, die meisten davon müssen durch den Nutzer korrigiert werden. Erst wenn die Validierung keine Fehler liefert, können die Änderungen am Dokument gespeichert werden.

Die meisten Datentyp-Fehler, die beim Bearbeiten der XML-Dokumente auftreten können, werden durch den Editor automatisch korrigiert. Dazu gehören z.B. Elemente, die nur Integer-Werte annehmen dürfen. Es ist zwar möglich, diese Fehler auch durch den Parser aufdecken und anschließend vom Editor beheben zu lassen, allerdings ist die Umsetzung deutlich aufwändiger und das Ergebnis wird dadurch nicht verändert.

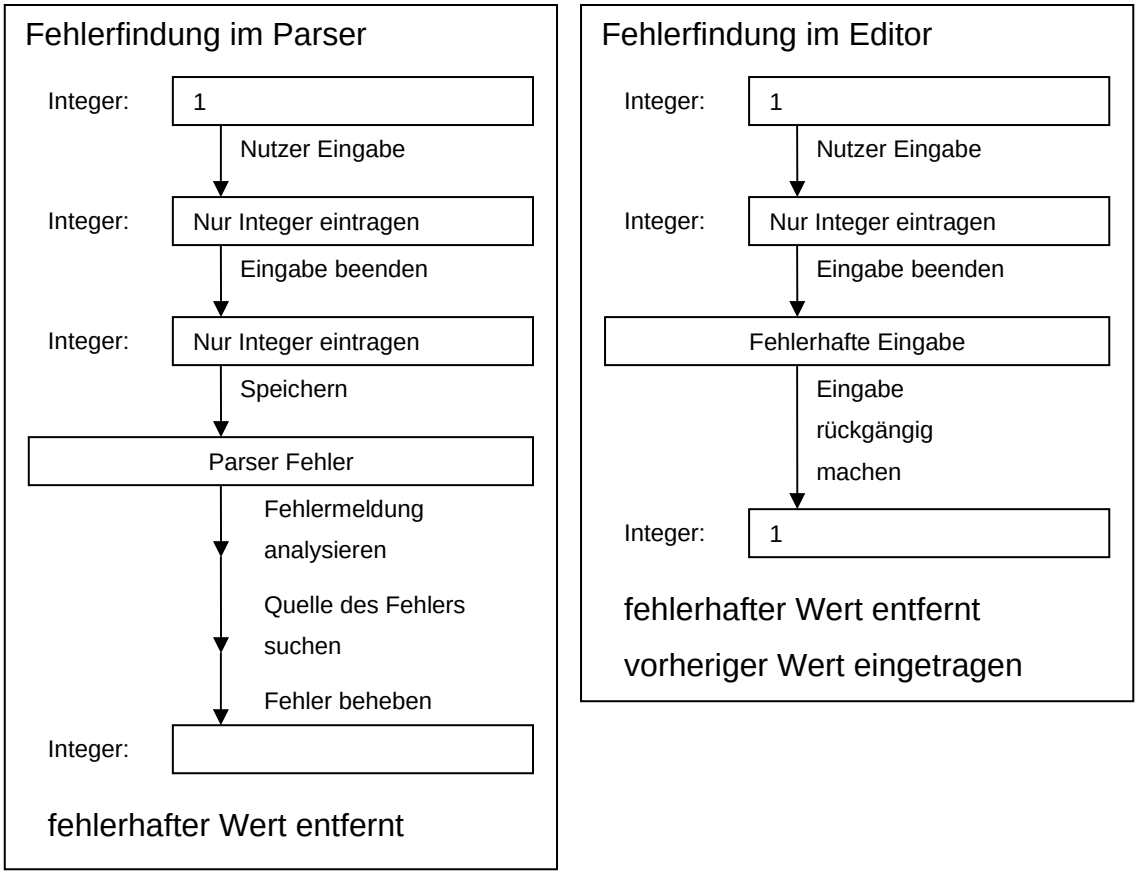


Abbildung 3.7 Vergleich der Fehlerbehebung falscher Datentypen bei Parser und GUI Code

Ebenso wie die Fehleranalyse und Fehlerbehebung werden Beschränkungen, die durch das Schema gegeben sind, wie z.B. eine minimale oder maximale Anzahl an Elementen, direkt durch den Editor beachtet. Wird z.B. ein Element erstellt, welches das festgelegte Minimum oder Maximum unter- bzw. überschreitet, so wird durch den Parser die fehlerhafte Anzahl erst nach dem Einfügen des Elements in das Dokument gemeldet. Das führt dazu, dass das überzählige Element wieder gelöscht werden muss. Wird dagegen die Anzahl der Elemente durch die Kontrollelemente des Editors überwacht, ist es möglich, die entsprechenden Funktionen zum Einfügen neuer Elemente über das Maximum, oder zum Löschen von Elementen unter dem Minimum, zu deaktivieren.

Auch wenn mit dem Parser dem Editor ein mächtiges Tool zur Fehlerfindung zur Verfügung steht, ist es doch in vielen Fällen einfacher diese Fehler durch den Editor zu verhindern, anstatt sie nach der Fehlermeldung zu korrigieren. Bei den Fehlern, die der Editor-Prototyp nicht verhindern bzw. nicht korrigieren kann - allen voran inhaltliche Fehler - ist der Nutzer gefragt, diese zu verhindern und zu korrigieren.

3.3 Überblick über die Klassen des Generators

3.3.1 Der Kernstück des Generators – die Klasse UIGenerator

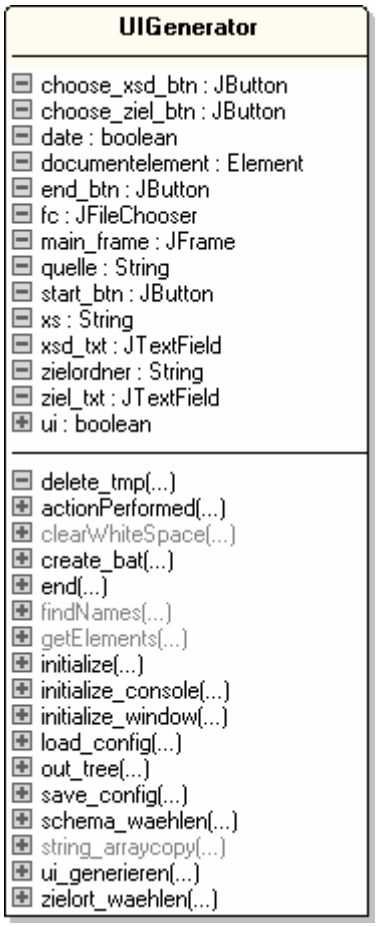


Abbildung 3.8 UML Klassendiagramm UIGenerator

Die Klasse *UIGenerator* enthält alle Variablen und Funktionen, die benötigt werden, um den Editor-Code zu generieren. Hierzu wird die *initialize*-Funktion durch die *main*-Klasse aufgerufen. Diese Funktion startet nun den Generator abhängig von den gegebenen Parametern. Standardmäßig wird der Generator als grafische Anwendung gestartet, sollte allerdings der Parameter „-c“ genutzt worden sein, wird der Generator als Konsolenanwendung initialisiert. Diese beiden Anwendungsarten unterscheiden sich nur in der Art der Eingabe und nicht in der Art der Verarbeitung.

Nach der Eingabe der Pfade für die Schema-Datei und für den Zielordner für die Generierung, wird die Funktion *ui_generieren* gestartet. Diese liest zuerst das Schema-Dokument ein und überträgt es in ein DOM. Anschließend ermittelt

der Generator eine Liste aller nach Schema möglichen Wurzelelemente. Dies entspricht einer Liste aller Kindelemente des Schema-Wurzelelements. Der Nutzer muss nun aus der Liste das Element wählen, welches in den auf dem Schema basierenden XML-Dokumenten das Wurzelelement ist. Anschließend wird noch die Angabe eines Präfix verlangt, welcher für die späteren Dateien verwendet wird.

An diesem Punkt beginnt nun der Generator mit der tiefgreifenden Analyse des Schemas. Dabei wird die Funktion *getElements* genutzt. Diese Rekursive Funktion durchläuft den gesamten Baum des DOM und speichert dessen Inhalt als *GenElemente* in einem neuen Baum, welcher dann für die weitere Generierung verwendet wird. Sobald der Baum vollständig analysiert ist, werden die einzelnen Klassen zur Generierung des Codes aufgerufen: *createMain*, *createView* und *createControl*. Dabei wird den Klassen jeweils der Baum übergeben.

Nachdem der Code generiert wurde, kopiert der Generator alle Dateien in den Zielordner und erstellt zwei Batch-Dateien über die Funktion *create_bat*. Der Generator speichert dabei auch die Pfade der XSD-Datei und des Zielordners. Diese werden beim nächsten Aufruf des Generators geladen und entsprechend eingetragen. Dadurch ist es bei mehrmaligem Generieren auf Basis einer bestimmten Schema-Datei nicht nötig, den Pfad neu zu ermitteln. Zuständig hierfür sind die Hilfsfunktionen *save_config* und *load_config*.

Die Klasse *UIGenerator* enthält noch eine Reihe weiterer Hilfsfunktionen. *schema_waehlen* und *zielort_waehlen* werden in der grafischen Anwendung jeweils für die Auswahl der XSD-Datei und des Zielordners genutzt.
findNames, *string_arraycopy* und *clearWhiteSpace* werden während der Analyse des Schemas genutzt. Die Funktion *clearWhiteSpace* wird dabei am häufigsten verwendet, da sie eine Schwäche des Parsers ausgleicht und das jeweilige Schema-Element von allen Whitespace-Kindelementen befreit.
Zum Kopieren der generierten Dateien werden 2 Hilfsklassen aufgerufen: *CopyFiles* und *CopyChanges*. *CopyFiles* dient dabei dem einfachen Kopieren von Dateien und Ordnern, während *CopyChanges* die in Kapitel 3.2.3 beschriebene Funktionalität zum Übernehmen von Änderungen bereitstellt.

3.3.2 Die Klasse zum Speichern der Schema-Daten – GenElement

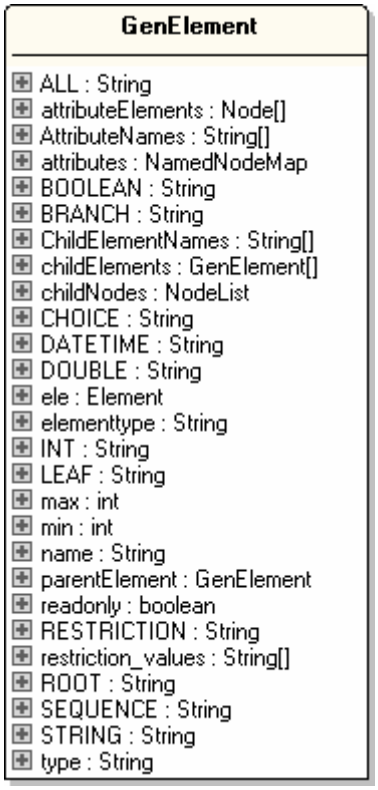


Abbildung 3.9 UML Klassendiagramm GenElement

Die Grundelemente der *GenElement*-Klasse wurden bereits in Kapitel 3.2.1 beschrieben. An dieser Stelle soll deshalb nur auf Erweiterungen eingegangen werden.

Die *GenElement*-Klasse enthält eine Reihe von Konstanten, welche für die Werte von *type* und *elementtype* verwendet werden.

ROOT, *BRANCH* und *LEAF* werden dabei für *type* verwendet und beschreiben die Funktion des entsprechenden *GenElements* im Baum.

BOOLEAN, *CHOICE*, *DATETIME*, *DOUBLE*, *INT*, *RESTRICTION*, *SEQUENCE* und *STRING* enthalten den jeweiligen Daten- oder Elementtyp für *elementtype*.

max und *min* enthalten jeweils die Maximal- und Minimalwerte des Elements. Diese Werte werden jeweils mit dem Standardwert 1 initialisiert; dies entspricht den Standardwerten im Schema.

readonly wird genutzt um anzugeben, ob ein Element als „*read only*“ deklariert wurde.

restrcition_values enthält die Enumerationswerte, die das Element annehmen kann. Es wird nur bei „*xs:restriction*“ Elementen genutzt.

3.4 Erläuterung der Herangehensweise und Lösung anhand des Generators und eines Fallbeispiels

3.4.1 Das Fallbeispiel, eine einfache Lagerverwaltung

3.4.1.1 Die Anforderungen

Um die Funktionsweise zu demonstrieren, soll hier eine einfache Lagerverwaltung als Beispiel dienen. Dieses Fallbeispiel wurde innerhalb der Professur Softwaretechnik der Hochschule Mittweida schon mehrfach bei Wissenschaftlichen Untersuchungen verwendet. Zum Beispiel in der Arbeit von Steffen Bönsch [Boensch 09].
Gegeben seien eine Reihe funktioneller Anforderungen, welche die Softwarelösung erfüllen muss, sowie das Datenmodell in Form von Klassendiagrammen.

```
Funktionelle Anforderungen
/F100/
Stammdatenpflege von Teilen und Fächern (Neu, Ändern, Löschen, Kopieren/Klonen)
/F200/
Einlagern einer Menge von Teilen gleicher Identnummer (z.B. Kreutzschlitzschraube M4 x 30) in ein bestimmtes Lagerfach. Gibt es das Teil noch nicht in den Stammdaten soll es bei dieser Gelegenheit auf Bedienerabfrage hin in den Stammdaten angelegt werden.
/F300/
Auslagern einer Menge von Teilen gleicher Identnummer aus einem bestimmten Fach (Das Suchen und Finden soll anhand Filterangaben schnell und effektiv erfolgen). Wenn der Bestand auf 0 geht, soll im Falle eines temporären Teiles (siehe Klassendiagramm) gefragt werden, ob der Teilestamm gleich mit gelöscht werden soll.
/F400/
Es soll möglich sein, eine Teilmenge/Gesamtmenge in ein anderes Lagerfach zu buchen.
/F500/
Es soll möglich sein, zu den Punkten /F200/-/F400/ Korrekturbuchungen (Rückgängig, Stornierung, Undo) durchzuführen.
/F600/
Das System soll das Ausführen der Funktionen /F100/-F500/ in jedem Falle protokollieren. Es ist zu beachten, dass die Protokolleintragungen auch überdauern, wenn temporäre Teile im Teilestamm wieder gelöscht werden.
```

Listing 3.3 Anforderungen Fallbeispiel Lagerverwaltung

Das gegebene Datenmodell besteht aus vier Entities (*Teil*, *Fach*, *Bestandskonto* und *Lagerbewegung*) und drei Einheitentypen (*BuchungsKategorieET*, *TeileKategorieET* und *FachKategorieET*). Teil und Fach beschreiben dabei die Stammdaten und werden durch Bestandskonto koordiniert.

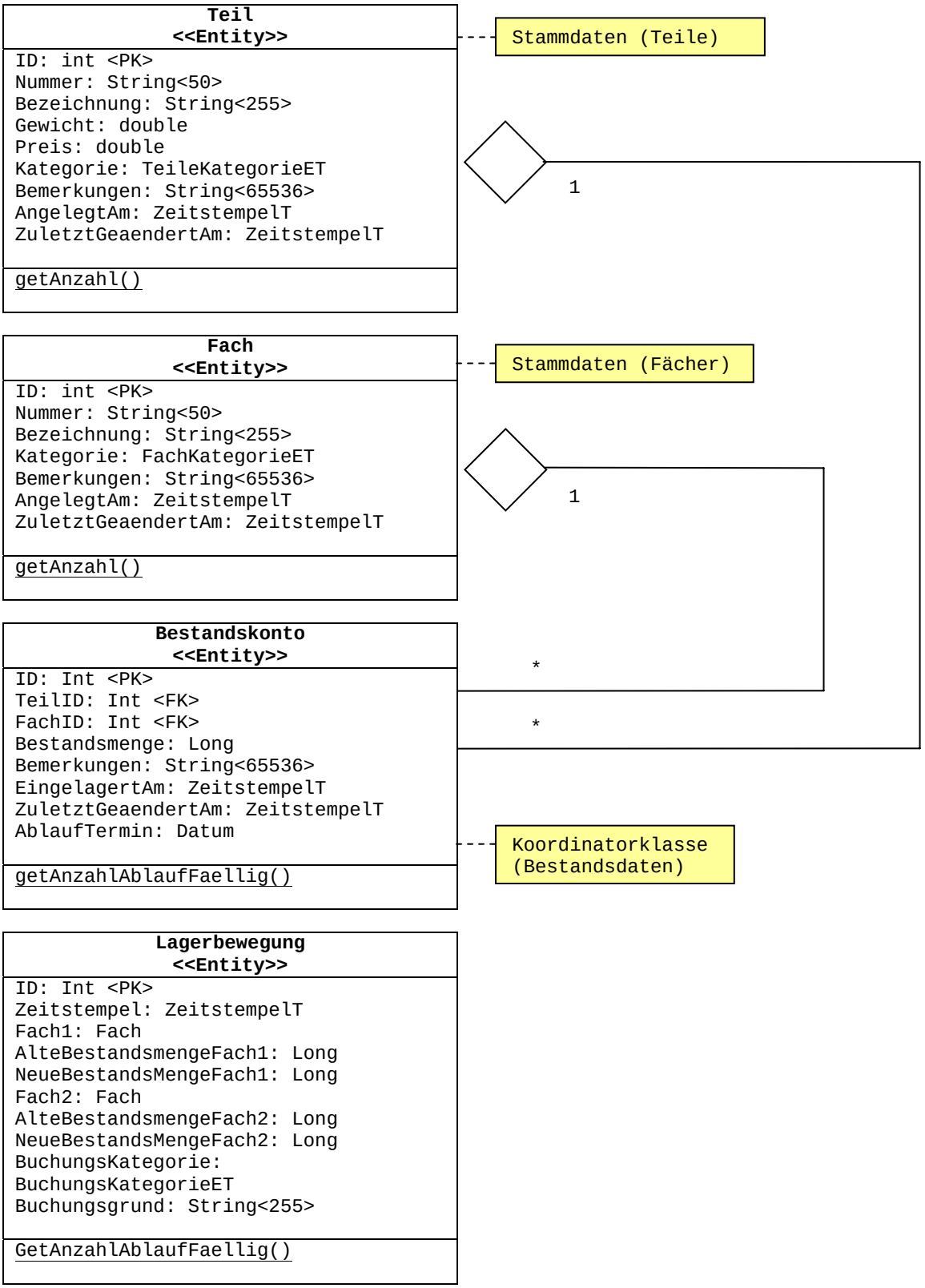


Abbildung 3.10 Klassendiagramm Fallbeispiel Lagerverwaltung

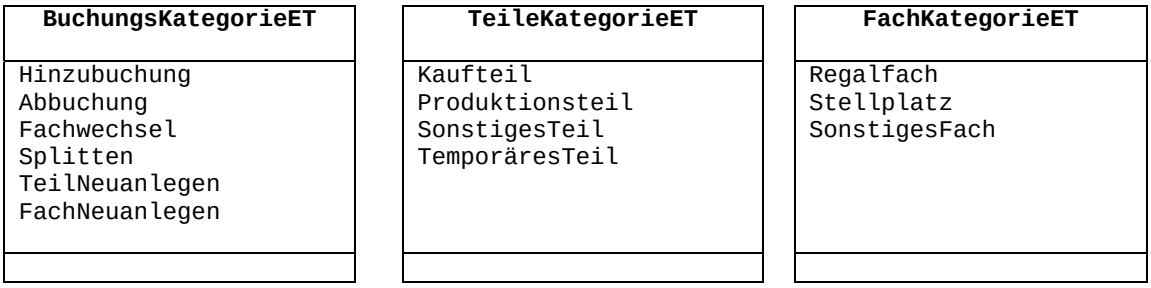


Abbildung 3.11 Fortsetzung Klassendiagramm Fallbeispiel Lagerverwaltung

3.4.1.2 Umsetzung des Datenmodell als XML-Schema

Der erste Schritt ist nun, die Klassendiagramme des Datenmodells in ein entsprechendes XML-Schema umzusetzen, welches dann vom Generator genutzt wird.

Aus dem Klassendiagramm ergibt sich folgendes Schema für die Stammdaten, das Bestandskonto sowie die Lagerbewegungen.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name=”lager”>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="StammdatenTeile">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Teil" type="Teil"
                   maxOccurs="unbounded" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="StammdatenFaecher">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Fach" type="Fach"
                   maxOccurs="unbounded" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Bestandskonten">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Bestandskonto" type="Bestandskonto"
                   maxOccurs="unbounded" minOccurs="0" />
```

```
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Lagerbewegungen">
     <xs:annotation>
      <xs:appinfo>readonly</xs:appinfo>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Lagerbewegung" type="Lagerbewegung"
                   maxOccurs="unbounded" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Listing 3.4 Schema Prototyp für das Fallbeispiel Lagerverwaltung

Es wäre auch möglich, die Daten direkt als Sequenz anzugeben und auf die hier verwendeten Stammdaten-Elemente zu verzichten, allerdings würde dies die Übersichtlichkeit der auf dem Schema basierenden XML-Dateien und des generierten GUIs stark beeinträchtigen.
Unter *StammdatenTeile* und *StammdatenFächer* werden alle *Teile* und *Fächer* angegeben.
Bestandskonto, im Datenmodell die Koordinatorklasse, beinhaltet im Schema alle Bestandsdaten, jedoch keine *Fächer* oder *Teile*.
Lagerbewegungen enthält alle vorgenommenen Lagerbewegungen. In der Software muss später sichergestellt werden, dass diese *Lagerbewegung*-Elemente automatisch generiert und gespeichert werden, da dies in den funktionellen Anforderungen vorgegeben ist.
Da in der XML-Datei nur die Daten gespeichert sind, muss die funktionelle Verbindung zwischen den Bestandskonten, den Stammdaten und den Lagerbewegungen später durch die Software übernommen werden.

Die einzelnen Entitäten des Datenmodells werden im Schema jeweils durch einen komplexen Typ dargestellt, welcher alle verschiedenen Werte als Elemente enthält.

Zum Beispiel hat der *complexType* für das Bestandskonto die folgende Form:

```
<xs:complexType name="Bestandskonto">
 <xs:sequence>
  <xs:element name="ID" type="xs:integer" />
  <xs:element name="TeilID" type="xs:integer" />
  <xs:element name="FachID" type="xs:integer" />
  <xs:element name="Bestandsmenge" type="xs:integer" />
  <xs:element name="Bemerkungen" type="xs:string" />
  <xs:element name="AngelegtAm" type="xs:datetime" />
  <xs:element name="ZuletztGeaendertAm" type="xs:datetime" />
  <xs:element name="AblaufTermin" type="xs:datetime" />
 </xs:sequence>
</xs:complexType>
```

Listing 3.5 Schema komplexe Typen für das Fallbeispiel Lagerverwaltung

Die Typen von *Teil* und *Fach* unterscheiden sich dabei nur durch die verschiedenen Elementnamen und Typen, die jeweils durch das Datenmodell vorgegeben sind.

Die *Lagerbewegung* dagegen beinhaltet neben den einfachen Typen, wie *String* oder *Integer*, auch zwei Elemente des Typs *Fach*, was später bei der Codegenerierung beachtet werden muss.

Die drei Kategorie-Einheiten-Typen werden jeweils als *simpleType* mit entsprechender Enumeration angegeben.

Als Beispiel soll hier der Typ *BuchungsKategorieET* dienen:

```
<xs:simpleType name="BuchungsKategorieET">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Hinzubuchung"/>
  <xs:enumeration value="Abbuchung"/>
  <xs:enumeration value="Fachwechsel"/>
  <xs:enumeration value="Splitten"/>
  <xs:enumeration value="TeilNeuanlegen"/>
  <xs:enumeration value="FachNeuanlegen"/>
 </xs:restriction>
</xs:simpleType>
```

Listing 3.6 Schema Kategorien für das Fallbeispiel Lagerverwaltung

3.4.2 Die GUI-Prototyp-Generierung

3.4.2.1 Die Analyse des XML-Schemas der Lagerverwaltung

Als erstes wird nun die XSD-Datei durch den Generator gelesen. Dabei wandelt er die Daten in eine Baumstruktur um, welche - in vereinfachter Form - dem Aufbau einer auf dem Schema basierten XML-Datei entspricht.
Die Baumstruktur besitzt die im folgenden gekürzt dargestellte Form.

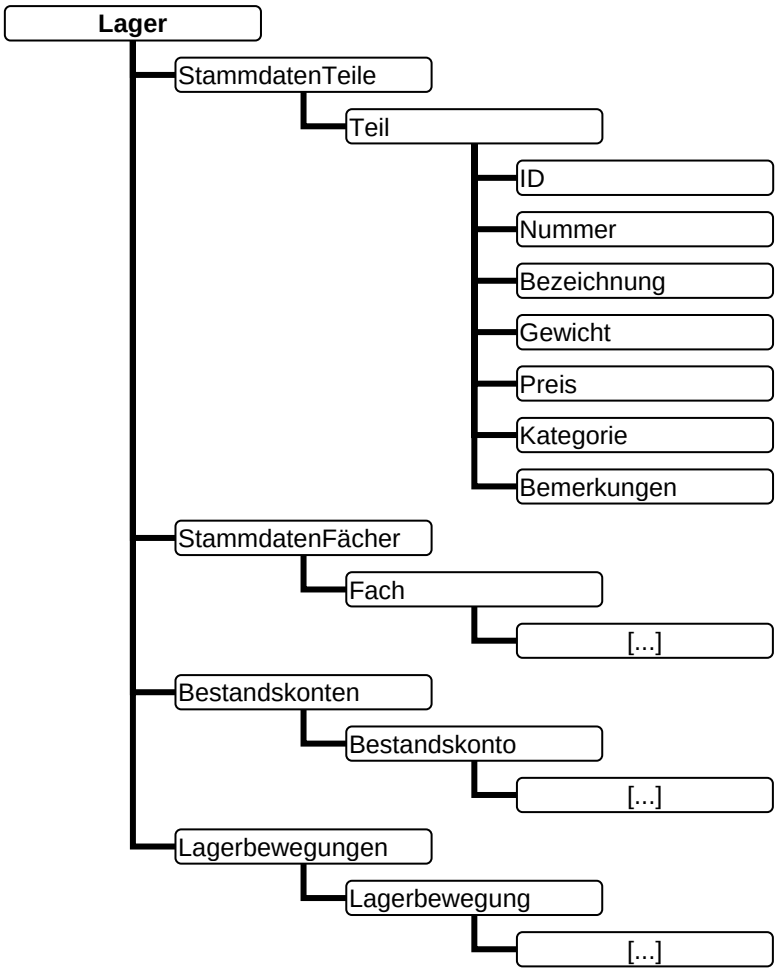


Abbildung 3.12 Baumdiagramm Schema Lagerverwaltung

Die Kindelemente von *Fach*, *Bestandskonto* und *Lagerbewegung* sind dabei analog zu denen von *Teil*, ihren im Schema gegebenen Daten entsprechend, im Baum enthalten.

3.4.2.2 Generierung des Prototypen

Nun da der Generator über alle wichtigen Daten verfügt, beginnt er mit der Generierung des Quellcodes der einzelnen Dateien: *Main*, *Control* und *View*.

Doch bevor diese erstellt werden, sind noch zwei Eingaben erforderlich. Zum einen ist die Angabe des Wurzelelements der später zu bearbeitenden XML-Dokumente notwendig. Dieses wählt man aus allen Kindelementen des Wurzelelements des Schemas. Im Falle der Lagerverwaltung sind dies: *Lager*, *Bestandskonto*, *Fach*, *Teil*, *Lagerbewegung*, *BuchungsKategorieET*, *TeileKategorieET* und *FachKategorieET*, wobei *Lager* das zu verwendende Element ist. Dies ist erforderlich, da die Reihenfolge der Kindelemente des Wurzelelements im Schema selbst keine Rolle spielt, wodurch nur der Nutzer sagen kann, welches Element später die Rolle des Wurzelelements übernimmt. Zum anderen wird vom Nutzer die Angabe eines Präfixes verlangt, der dann für die Dateinamen sowie mehrere Funktionsnamen verwendet wird - Standard hierfür ist der Name des Wurzelelements.

Die Code-Generierung beginnt bei der *Main*-Klasse, *LagerMain*. Die *Main*-Klasse enthält die meisten allgemeinen Funktionen, welche nicht oder nur geringfügig an das jeweilige Schema angepasst werden müssen. Die Startfunktionen und Funktionen zum Speichern von Änderungen sind dabei unabhängig vom Schema und bei jedem generierten Code gleich. Funktionen zum Erstellen neuer Elemente, Bearbeiten existierender Elemente und zum Erstellen des Baumes aus der XML, welcher später im GUI dargestellt wird, sind stark vom Schema abhängig und werden dementsprechend stark angepasst.
Hier ein Beispiel (Die Rot markierten Codefragmente wurden speziell angepasst, der Rest ist in genau dieser Form in jedem generierten Code enthalten.):

```
//Globale Variablen
private Document doc;
protected Element de;

protected LagerView Lager_view = new LagerView();
protected LagerControl Lager_ctrl = new LagerControl();
```

```
[...]

public void Lager_parse(String url) {
  try {
   parser.setFeature("http://xml.org/sax/features/validation", true);
   parser.setFeature("http://apache.org/xml/features/validation/schema",true);
   parser.setFeature("http://apache.org/xml/features/dom/include-ignorable-
whitespace", false);
  }
  catch (SAXNotRecognizedException e) { e.printStackTrace(); }
  catch (SAXNotSupportedException e){ e.printStackTrace(); }

  try {
    parser.parse(url);
  } catch (SAXException e) {
    JOptionPane.showMessageDialog(new JFrame(),
                                  "Fehler in der xml Datei\n",
                                  "Parser Fehler",
                                  JOptionPane.ERROR_MESSAGE);
    System.exit(0);
  }
  catch (IOException e) {
    JOptionPane.showMessageDialog(new JFrame(),
                                    "Fehler beim lesen der Datei\n",
                                    "Parser Fehler",
                                    JOptionPane.ERROR_MESSAGE);
    System.exit(0);
  }
  doc = parser.getDocument();
  de = doc.getDocumentElement();

  Lager_ctrl.Lager_view = Lager_view;
  Lager_ctrl.Lager_main = this;

  Lager_view.newTree(initTreeRoot());
  Lager_view.tree.addTreeSelectionListener(Lager_ctrl);

  Lager_view.neues_element_btn.addActionListener(Lager_ctrl);
  Lager_view.loeschen_btn.addActionListener(Lager_ctrl);
  Lager_view.aendern_btn.addActionListener(Lager_ctrl);

  Lager_view.frame.setVisible(true);
}
[...]
```

Listing 3.7 Codebeispiel LagerMain

Wenn das Erstellen der *Main*-Klasse erfolgreich abgeschlossen ist, beginnt die Generierung der *Control*-Klasse, *LagerControl*. Diese beinhaltet den *Action-Listener* sowie den *TreeSelectionListener* für das GUI. Dabei werden beide entsprechend für die im GUI enthaltenen GUI Komponenten erstellt.
Der folgende Codeausschnitt bezieht sich auf das *ActionEvent*, das durch den späteren *„Neues Element einfügen“*-Knopf ausgelöst wird.

```
//Globale Variablen
protected LagerMain Lager_main;
protected LagerView Lager_view;
[...]
public void actionPerformed(ActionEvent e){
  String command = e.getActionCommand();
[...]
  if(command.equals("neu")){
    Lager_view.loeschen_btn.setEnabled(false);

    JFrame frame = new JFrame();
    Object[] types = { "Teil", "Fach", "Bestandskonto" };
    String s = (String)JOptionPane.showInputDialog(frame,
                                                   "Typ des neuen Elements",
                                                   "Elementtyp wählen",
                                                   JOptionPane.PLAIN_MESSAGE,
                                                   null,
                                                   types,
                                                   "");
    if ((s != null) && (s.length() > 0)) {
      Lager_main.neues_element(s);
      Lager_view.tree.removeTreeSelectionListener(this);
      Lager_view.newTree(Lager_main.initTreeRoot());
      Lager_view.tree.addTreeSelectionListener(this);
      Lager_view.editscrollPane.setViewportView(null);
      Lager_view.atteditscrollPane.setViewportView(null);
    }
  }
[...]
```

Listing 3.8 Codebeispiel LagerControl

Zu beachten ist dabei, dass bereits in der Abfrage nur jene Elemente angeboten werden, welche - durch das Schema vorgegeben - auch in der Lage sind mehr als ein Kindelement zu enthalten, bzw. welche nicht *„read only“* sind.

Die *View*, in Form von *LagerView*, ist die Klasse, welche am umfangreichsten an das Schema angepasst wird. Da durch sie die Komponenten des GUI definiert werden, ist lediglich die Struktur immer dieselbe, während sich der Code meist komplett unterscheidet - mit Ausnahme einer einzigen Funktion zum Initialisieren der Baum-Darstellung. Innerhalb der *View* wird für jedes Element ein Eigenes JPanel definiert, welches die jeweils auf das Element zugeschnittenen Eingabefelder enthält. Bei der Auswahl eines Elements im Baum wird das entsprechende JPanel angezeigt, während die anderen derzeit nicht benötigten im Hintergrund bleiben. Das führt zwar zu einer sehr umfangreichen Komponenten-Sammlung, allerdings ist das einfache Austauschen einer GUI-Komponente in Form eines JPanels weniger rechenaufwändig als das komplette Neustrukturieren der Eingabefelder abhängig vom aktuell angewählten Element.
Als Beispiel alle mit dem Element „Teil“ verbunden Komponenten.

```
[...]
public JPanel Teil_editpanel;
public JPanel Teil_att_editpanel;
[...]
public JLabel Teil_ID_lbl = new JLabel("ID");
public NumberFormat Teil_ID_format = NumberFormat.getNumberInstance();
public JFormattedTextField Teil_ID_txt;
public JLabel Teil_Nummer_lbl = new JLabel("Nummer");
public JTextField Teil_Nummer_txt;
public JLabel Teil_Bezeichnung_lbl = new JLabel("Bezeichnung");
public JTextField Teil_Bezeichnung_txt;
public JLabel Teil_Gewicht_lbl = new JLabel("Gewicht");
public NumberFormat Teil_Gewicht_format = NumberFormat.getNumberInstance();
public JFormattedTextField Teil_Gewicht_txt;
public JLabel Teil_Preis_lbl = new JLabel("Preis");
public NumberFormat Teil_Preis_format = NumberFormat.getNumberInstance();
public JFormattedTextField Teil_Preis_txt;
public JLabel Teil_Kategorie_lbl = new JLabel("Kategorie");
public String[] Teil_Kategorie_choices = { "Kaufteil", "Produktionsteil",
                                           "SonstigesTeil", "TemporaeresTeil"
};
public JComboBox Teil_Kategorie_combo;
public JLabel Teil_Bemerkungen_lbl = new JLabel("Bemerkungen");
public JTextField Teil_Bemerkungen_txt;
public JLabel Teil_AngelegtAm_lbl = new JLabel("AngelegtAm");
public JPanel Teil_AngelegtAm_panel;
public JDateChooser Teil_AngelegtAm_datechooser =
```

```
                new JDateChooser("dd.MM.yyyy  HH:mm:ss","##.##.####
##:##:##",'_');
public JLabel Teil_ZuletztGeaendertAm_lbl = new JLabel("ZuletztGeaendertAm");
public JPanel Teil_ZuletztGeaendertAm_panel;
public JDateChooser Teil_ZuletztGeaendertAm_datechooser =
                new JDateChooser("dd.MM.yyyy  HH:mm:ss","##.##.####
##:##:##",'_');
[...]
```

Listing 3.9 Codebeispiel LagerView

Nach der Fertigstellung der *View* wendet sich der Generator nun dem Zielordner zu. Dieser wird erstellt und die gerade generierten *LagerMain*, *LagerControl* und *LagerView* hineinkopiert, gefolgt von den benötigten Bibliotheken - beispielsweise für das *JDateChooser* Element. Sollte der Ordner und sein Inhalt bereits vorhanden sein, bietet der Generator an, die vorhandenen Dateien zu überschreiben oder die Änderungen zu übernehmen, welche seit dem letzten Generierungs-Vorgang hinzugefügt wurden. Der Editor muss nun nur noch kompiliert werden, um ihn in Betrieb nehmen zu können.

3.4.3 Analyse des generierten Prototypen anhand der Anforderungen des Fallbeispiels

3.4.3.1 Was kann der Prototyp?

Nachdem der Editor erfolgreich kompiliert wurde, ist es an der Zeit, ein XML-Dokument, welches auf dem Schema basiert, zu öffnen. Der Parser wird dabei so eingestellt, dass er die Fehlermeldungen in der Konsole ausgibt, aber das Dokument dennoch einliest, sofern die Fehler das Einlesen nicht unmöglich machen.

Nach der Auswahl des Dokuments wird nun das Editor-GUI geöffnet. Hier wurde der Editor mit einem Beispieldokument, welches ein Teil, zwei Fächer, ein Bestandskonto und eine Lagerbewegung enthält, gestartet.

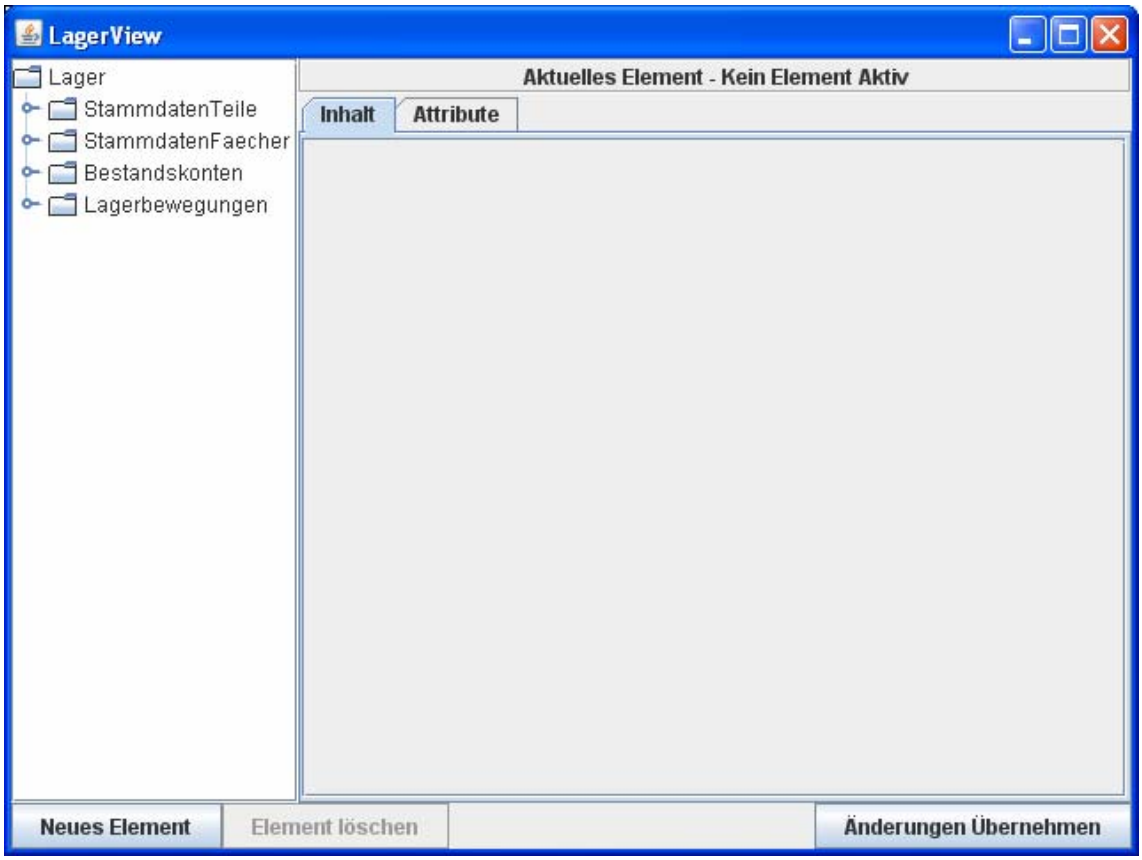


Abbildung 3.13 Lagerverwaltung GUI Prototyp Ansicht 1

Die Bezeichnung der Elemente des Baums auf der linken Seite des GUIs sind hier noch die durch den Generator gegebenen Standardwerte. Als Standard-wert wird der Inhalt des ersten Kindelements des jeweiligen Elements genutzt.

Im Falle aller hier enthaltenen Elemente ist dies das Element „ID“. Der Löschen-Knopf ist hier noch deaktiviert, weil er die Auswahl eines Elementes erfordert. In der rechten Hälfte des GUI werden bei Auswahl eines Elements die entsprechenden Eingabefelder angezeigt. Wenn kein Element ausgewählt ist, wird sie entsprechend leer dargestellt.

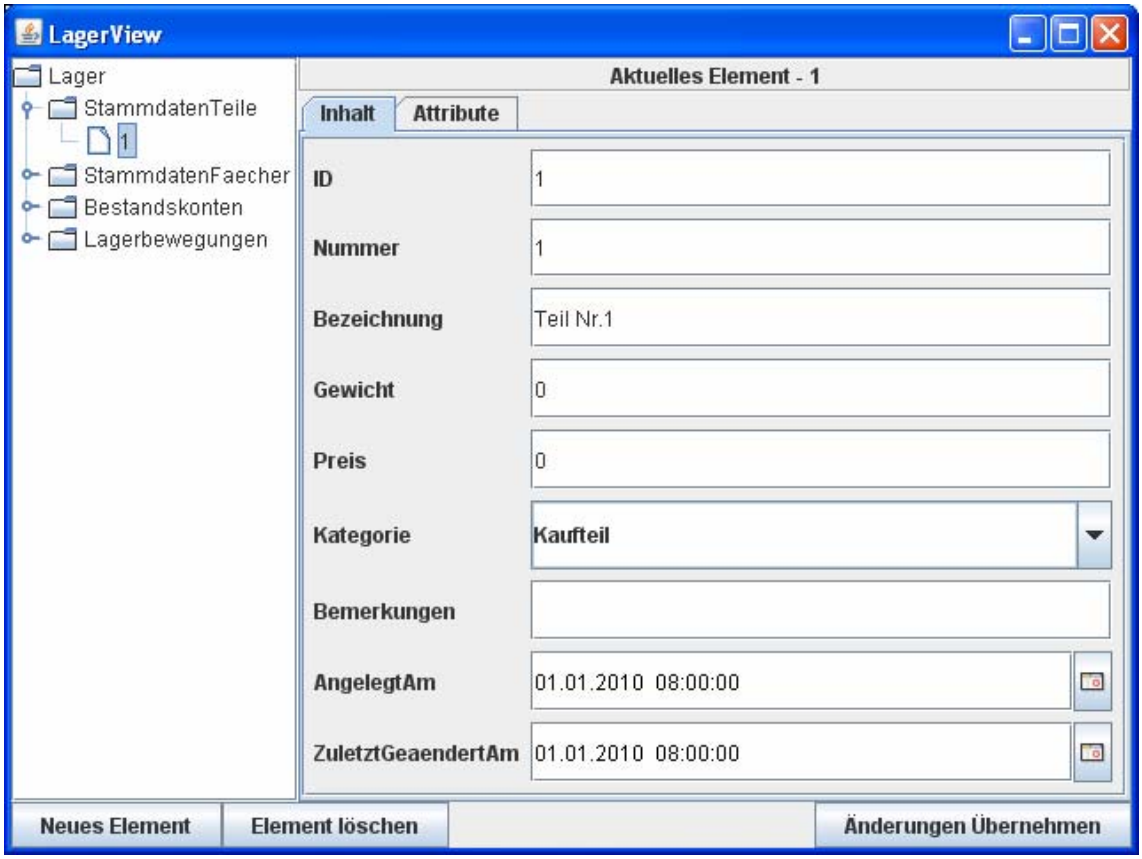


Abbildung 3.14 Lagerverwaltung GUI Prototyp Ansicht 2

Wie in der Abbildung zu sehen, wurde im Baum das „Teil“-Element angewählt, was zur Folge hat, dass auf der rechten Hälfte des GUI nun die Eingabefelder angezeigt werden. Die Größe der Felder wird dabei durch das variable Layout automatisch an die Größe des GUIs angepasst.

Zu beachten ist, dass Änderungen an den Inhalten der Eingabefelder erst durch Druck des „Ändern“-Buttons in die jeweilige XML-Dateien übernommen werden.

3.4.3.2 Was muss der Prototyp noch können

Betrachtet man nun die funktionellen Anforderungen des Fallbeispiels, ist zu sehen, dass der Prototyp bereits die Lösung für einen Teil davon bietet.

```
/F100/
Stammdatenpflege von Teilen und Fächern (Neu, Ändern, Löschen, Kopie-
ren/Klonen)
```

Der Prototyp ermöglicht das Anlegen, Ändern und Löschen von Stammdaten. Kopieren/Klonen ist aufbauend auf diesen Funktionen durch das Hinzufügen eines entsprechenden Buttons zu erreichen.

```
/F200/
Einlagern einer Menge von Teilen gleicher Identnummer (z.B. Kreutzschlitz-
schraube M4 x 30) in ein bestimmtes Lagerfach. Gibt es das Teil noch nicht in
den Stammdaten soll es bei dieser Gelegenheit auf Bedienerabfrage hin in den
Stammdaten angelegt werden.
```

Durch Bearbeiten der *Bestandskonto*-Elemente ist es bereits möglich, diese Operation durchzuführen. Der Anforderung besser gerecht würde hier allerdings eine angepasste Eingabemaske.

```
/F300/
Auslagern einer Menge von Teilen gleicher Identnummer aus einem bestimmten
Fach (Das Suchen und Finden soll anhand Filterangaben schnell und effektiv
erfolgen). Wenn der Bestand auf 0 geht, soll im Falle eines temporären Teiles
(siehe Klassendiagramm) gefragt werden, ob der Teilestamm gleich mit gelöscht
werden soll.
```

Das Auslagern an sich ist erneut durch das simple Bearbeiten des *Bestandskontos* möglich, aber auch hier wäre eine entsprechende Eingabemaske vorteilhafter. Die Filter- und Suchfunktionen müssen noch erstellt werden. Die Abfrage der temporären Objekte nach jeder Aktion ist ebenfalls nicht enthalten.

```
/F400/
Es soll möglich sein, eine Teilmenge/Gesamtmenge in ein anderes Lagerfach zu
buchen.
```

Diese Anforderung ist durch die gegebenen Eingabemöglichkeiten ebenfalls teilweise erfüllt. Allerdings ist auch hier eine angepasste Eingabemaske nötig.

```
/F500/
Es soll möglich sein, zu den Punkten /F200/-/F400/ Korrekturbuchungen (Rück-
gängig, Stornierung, Undo) durchzuführen.
/F600/
Das System soll das Ausführen der Funktionen /F100/-F500/ in jedem Falle pro-
tokollieren. Es ist zu beachten, dass die Protokolleintragungen auch überdau-
ern, wenn temporäre Teile im Teilestamm wieder gelöscht werden.
```

Die Korrekturbuchungen selbst bauen auf den Protokollen auf, da sie diese nutzen können, um bestimmte Aktionen rückgängig zu machen.
Die Protokolle in Form von *Lagerbewegungen* sind durch ihre Definition als *readonly* im Schema nicht veränderbar; eine entsprechende Funktion, sie bei jeder Aktion zu erstellen, muss dem Editor hinzugefügt werden.

Zu diesen noch zu erfüllenden Anforderungen sind einige Darstellungsformen noch weiter anzupassen, z.B. mit welcher Bezeichnung die Baum-Elemente dargestellt werden sollen.

3.4.3.3 Nutzen des Prototyps für das Fallbeispiel

Wie im vorherigen Kapitel gezeigt, sind bereits einige Anforderungen des Fallbeispiels erfüllt. Die Vervollständigung der anderen Anforderungen wird durch den Prototypen ebenfalls begünstigt, da er die Grundlage für viele Funktionen und Masken bereits bereitstellt. Die Umsetzung des Datenmodells als XML-Schema hat zudem den Vorteil, dass man schnell eine XML-Datenbank samt Eingabe-GUI in Form eines einfachen Editors zur Verfügung hat.

4 Möglichkeiten die der Prototyp bietet

4.1 Welchen Nutzen bringt der Generator

Der Generator ermöglicht es schnell einen spezialisierten GUI-basierten XML-Editor zu erstellen, welcher im weiteren Verlauf der OOA und Softwareentwicklung erweitert werden kann. Dieser Editor-Prototyp wiederum bietet eine Vielzahl von Möglichkeiten, welche vor allem aus seiner vollständigen Anpassbarkeit hervorgeht. Er kann als einfacher Editor für ein XML-Datenmodell genutzt werden, er kann als Grundlage für eine entsprechende Bearbeitungs- oder Verwaltungssoftware - wie das Fallbeispiel Lagerverwaltung - genutzt werden, er kann aber auch problemlos in eine größere Software eingebaut werden und dort die Funktion eines Editors liefern.

Der größte Vorteil des Generators, auch gegenüber anderen Generatoren, besteht darin, dass er einen frei konfigurier- und bearbeitbaren Code liefert, welcher nach seinem Erstellen vollkommen unabhängig vom Generator arbeitet.

4.2 Wo liegen die Grenzen des Generators

Der Nachteil des Generators ist, dass er ein bereits vorhandenes XSD- und XML-Datenmodell voraussetzt. Ebenso muss das gelieferte Schema bestimmten Anforderungen entsprechen, die unter umständen eine Anpassung des Schemas verlangen.

Ebenso ist der generierte Editor nur ein Prototyp der zwar für einfache Bearbeitungen genutzt werden kann, aber nicht über die Leistungsfähigkeit eines spezialisierten, nicht anpassbaren XML-Editors verfügt.

Die größte Schwäche des Generator-Prototypen liegt vor allem in der Schema-Erkennung. Die große Menge an Möglichkeiten, welche durch XML-Schema gegeben sind, vollständig abzudecken würde dazu führen, dass der Generator deutlich umfangreicher ausfallen würde als er es bereits ist. Deshalb ist es notwendig, dass die für den Generator genutzten Schema-Dokumente für den Generator angepasst sind. So ist es z.B. nicht möglich Namespaces und mehrere Dokumente umspannende Schemas zu benutzen.

5 Zusammenfassung

5.1 Erreichte Ergebnisse

Das Ziel der Arbeit, die Grundlagen und Probleme der Verarbeitung mit XML und XML-Schema zu erläutern, wurde im Rahmen der Möglichkeiten einer Diplomarbeit erfüllt. Die Möglichkeiten, die XML bietet, sind zwar weitaus vielfältiger als hier beschrieben, allerdings würde die Beschreibung und Umsetzung all dieser Möglichkeiten und Lösungen den Rahmen der Arbeit sprengen. Dies betrifft sowohl die Möglichkeiten der XML-Dokumente und XML-Schemas, als auch die der Verarbeitung.

Das zweite Ziel der Arbeit war es, einen XML-Schema-basierten Java-Swing-GUI-Generator zu entwickeln. Dabei hat sich die Arbeit mit einigen ausgewählten Problemen der Verarbeitung und Entwicklung und deren Lösungen beschäftigt. Dabei ist ein Generator entstanden, der in der Lage ist, einen effektiven Editor mit einem leicht zu bedienenden GUI zu erstellen. Der Generator ermöglicht damit einen schnellen Einstieg in die Implementierung von Software, die auf einem XML-basierten Datenmodell aufbaut.
Die große Menge an Möglichkeiten, die XML und XSD selbst bieten, hat letztendlich dazu geführt, dass der Generator in seiner eigenen Leistungsfähigkeit beschränkt ist. So kann er nur dann fehlerfrei funktionieren, wenn das gelieferte Schema auf individuelle Namespaces verzichtet und sich vollständig in einer einzigen Datei befindet. Ebenso kann der Generator nicht alle Schema-Elemente vollständig und fehlerfrei verarbeiten. Da das Hauptziel des Editors darin bestand, einen schnellen Zugang und eine einfache Bearbeitung eines XML-basierten Datenmodells, welches im Verlauf einer OOA entwickelt wurde, zu ermöglichen, sind diese Beschränkungen vertretbar.

Zusammenfassend lässt sich sagen, dass die Arbeit die Grundlagen und den auf den Grundlagen basierenden Generator der Aufgabe entsprechend umsetzt, aber auch viel Platz für Erweiterungen und Vertiefungen des Themas bietet.

5.2 Ausblick

Der Generator selbst ist nur ein Grundstein, welcher noch viele Möglichkeiten der Erweiterung bietet, zum einen das Hinzufügen von bisher nicht erkannten Elementen, zum anderen die Unterstützung für Namespaces und über mehrere Dateien führende XSD- und XML-Dokumente.

Mit ein wenig mehr Aufwand wäre es auch möglich, den Generator in andere Programmiersprachen wie z.B. C# zu übertragen. Hierzu wäre es nötig, die zu generierenden Codefragmente sowie, abhängig von der jeweiligen Sprache, die Struktur der zu generierenden Dateien umzustellen.

Ebenso wäre es möglich, den Generator in eine Entwicklungsumgebung einzubinden, zum Beispiel als Plugin für das EMF – das Eclipse Modelling Framework. Dadurch wäre es möglich, sofern man Eclipse oder eine andere entsprechende Entwicklungsumgebung nutzt, den Generator noch stärker in den Entwicklungs- und Implementierungsprozess einzubinden.

Die in der Arbeit beschriebenen Grundlagen können auch für andere Generatoren, die sich nicht auf einen XML-Editor mit entsprechendem GUI beschränken, genutzt werden. Die Möglichkeiten, die XML und XML-Schema als Grundlage der Codegenerierung bieten, sind nahezu endlos und sie alle in einer einzigen Arbeit zu behandeln bzw. umzusetzen, ist nicht möglich.

6 Anhang

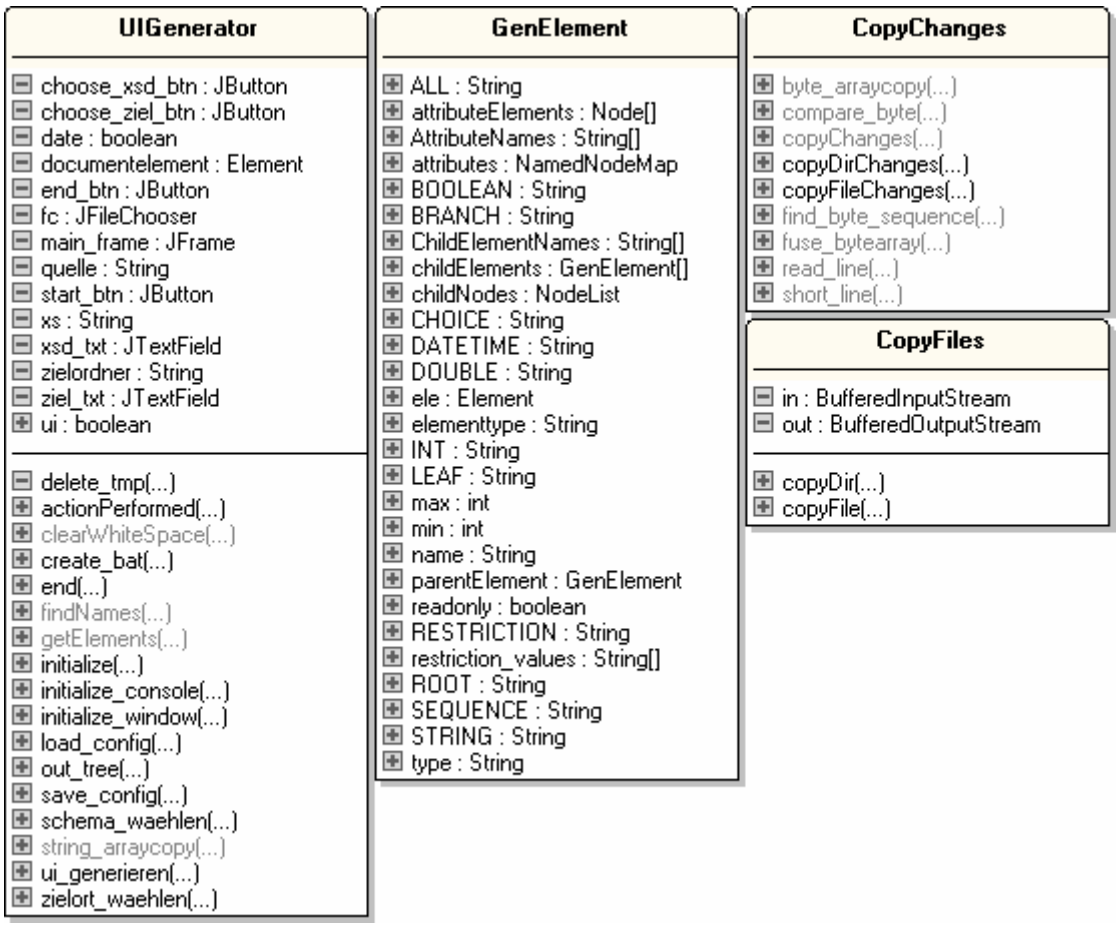


Abbildung 6.1 Anhang UIGenerator UML Klassendiagramm

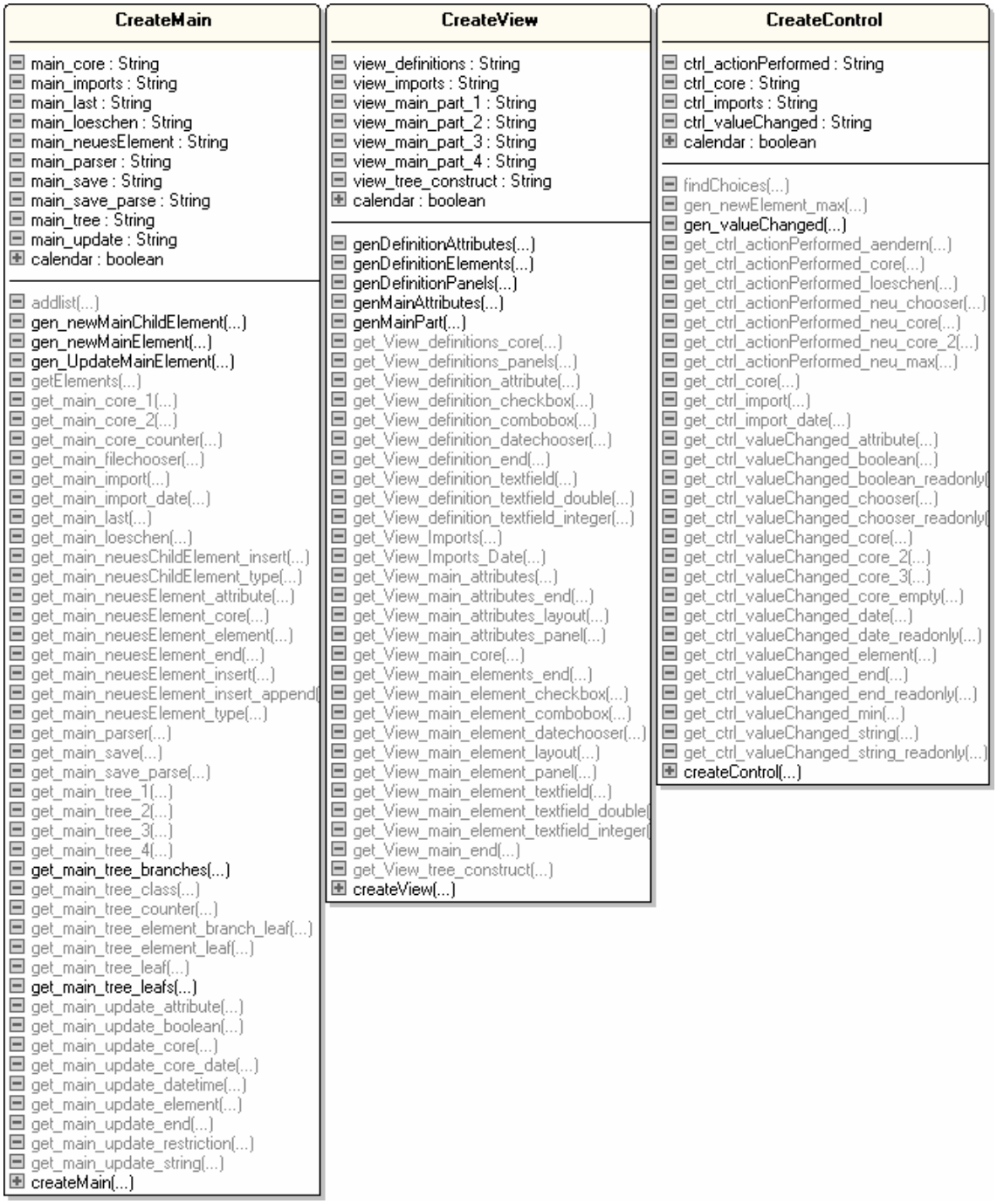


Abbildung 6.2 Anhang Fortsetzung UIGenerator UML Klassendiagramm

Die hier gezeigten Klassendiagramme wurden mithilfe von ESS-Model erstellt.

Eldean AB: ESS_Model. http://essmodel.sourceforge.net

Anleitung für die Benutzung des Code-Generators

Im Verzeichnis des UIGenerators befinden sich, wie im folgenden Bild zu sehen, 3 Dateien sowie 3 Verzeichnisse.

Die Quelldateien befinden sich im *src*-Verzeichnis bzw. dessen Unterverzeichnissen. Im *bin*-Verzeichnis befindet sich der kompilierte Quellcode und im *lib*-Verzeichnis liegen die benötigten Bibliotheken.
Die Datei *uigen.config* entsteht nach dem ersten Ausführen des Generators und enthält die Pfade des letzten erfolgreichen Generier-Vorgangs.
uigenerator starten bzw. *uigenerator.bat* dienen zum Starten des Generators.
Damit der Generator gestartet werden kann, ist Java in der Version 1.6 nötig.

Anforderungen an das XML-Schema

Es gibt eine Reihe von Anforderungen, die das Schema erfüllen muss, damit der Generator damit umgehen kann:

1. Es muss auf der W3C Schema Sprache basieren.
2. Es darf keine Namespaces enthalten.
3. Es darf nicht über mehrere Dateien verteilt sein.

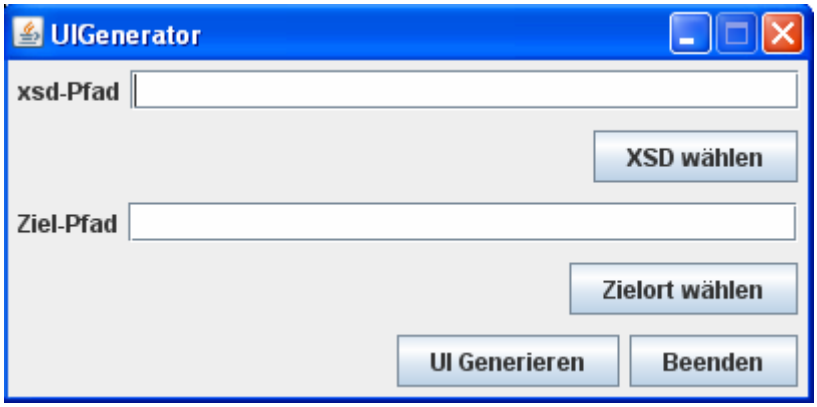
Es gibt einige Elemente, die nicht oder nur teilweise unterstützt werden. Es handelt sich hierbei hauptsächlich um selten verwendete Elemente und Elemente, die für Datenstrukturen eher ungeeignet sind. Eine vollständige Liste findet sich in der Dokumentation auf der der Arbeit beiliegenden CD.

Bedienung des Generators

Zum Starten des Generators genügt das Ausführen der *uigenerator.bat*-Datei. Diese enthält alle Kommandos die zum Starten nötig sind. Ebenso wird eine Konsole geöffnet, über welche während der Generierung zusätzliche Informationen ausgegeben werden.

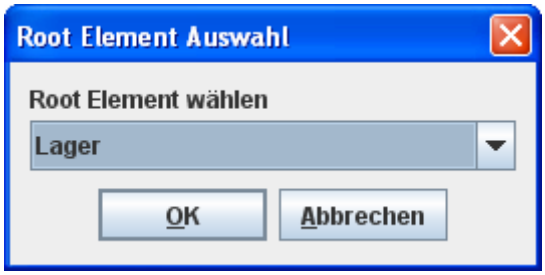
Sollte der Generator nicht starten bzw. eine Fehlermeldung erscheinen, dass der Befehl nicht gefunden werden konnte, ist dies darauf zurückzuführen, dass Java nicht installiert bzw. nicht im Classpath des Betriebssystems registriert ist.

Wenn der Generator erfolgreich gestartet wurde, ist nun folgendes Eingabefenster zu sehen:



Im Oberen Textfeld gibt man den Pfad der Schema-Datei an. Über den entsprechenden Knopf gelangt man zu einem Dateiauswahlfenster, welches die Auswahl erleichtert.
Das zweite Feld funktioniert analog zum ersten, allerdings wird hier der Pfad des Zielordners, in den der generierte Editor ausgegeben werden soll, angegeben.
Mit einem Druck auf *UI Generieren* startet der Generier-Vorgang.

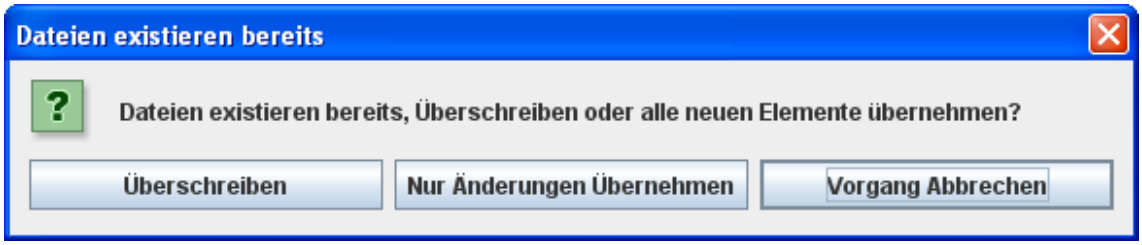
Während der Generierung erscheinen noch zwei weitere Eingabefenster.
Die Inhalte der beiden Auswahlfenster stammen hier aus dem Fallbeispiel aus Kapitel 3.4.



Hier wählt man das Element des Schemas, welches als Wurzelelement des XML-Dokumentes dient.

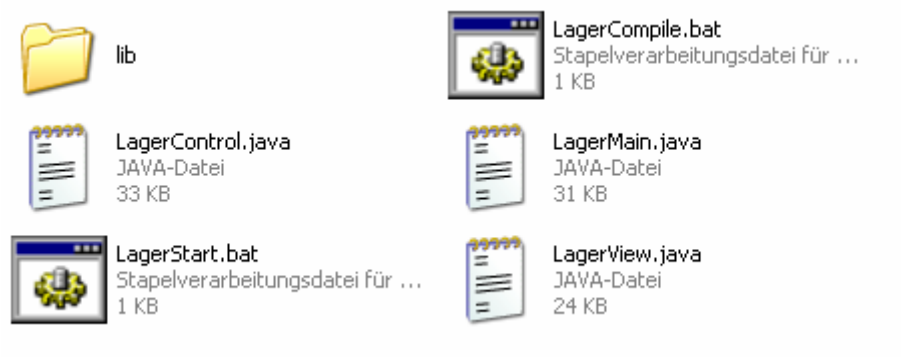
Hier wählt man den Datei-Präfix, welcher für die generierten Dateien verwendet wird.

Sollten der Zielordner und mindestens eine der zu generierenden Dateien bereits existieren, erscheint folgendes Auswahlfenster:



Überschreiben löscht die alten Dateien und kopiert die neuen in den Ordner.
Nur Änderungen übernehmen vergleicht die neuen Dateien mit den bereits vorhandenen und schreibt alle Unterschiede an der entsprechenden Stelle in die vorhandenen Dateien.
Vorgang Abbrechen beendet den Generator an dieser Stelle.
Nach dem erfolgreichen Kopieren wird der Generator automatisch beendet.

Im angegebenen Zielordner befinden sich nun die folgenden Dateien:



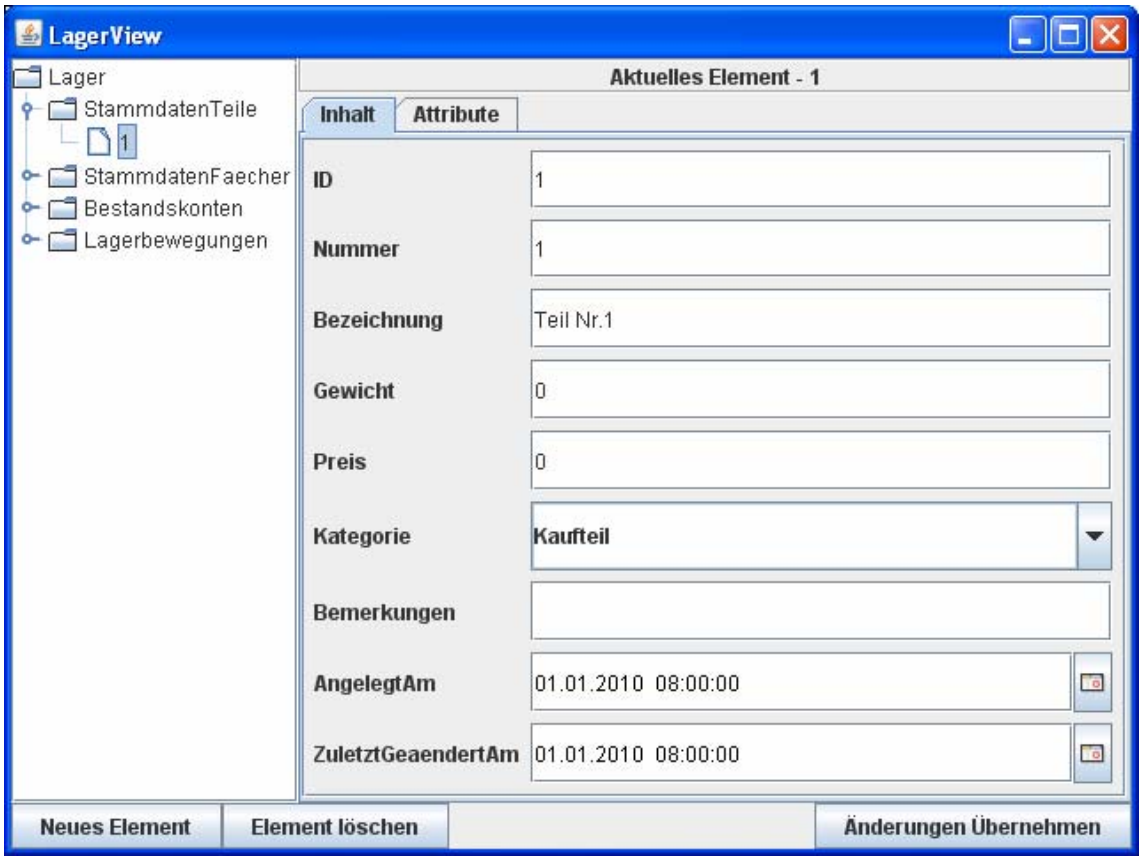
lib enthält die Bibliotheken, welche durch den Editor genutzt werden, zum Beispiel Xerces.

LagerMain.java, *LagerView.java* und *LagerControl.java* enthalten den Editor-Code.

LagerCompile.bat enthält alle Befehle, die nötig sind, um den Editor sofort zu kompilieren.

Analog dazu enhält *LagerStart.bat* alle Befehle zum Starten des kompilierten Editors.

Das GUI des generierten Editors hat das folgende Aussehen; der Inhalt stammt aus dem Fallbeispiel des Kapitels 3.4:



Zur Auswahl eines Elementes genügt es, das entsprechende Element im Baum auf der linken Hälfte des GUI zu markieren. Die Anzeige der Eingabefelder auf der rechten Hälfte wird sich entsprechend ändern.

Die Text- bzw. Eingabefelder sind entsprechend ihren im Schema gegebenen Datentypen formatiert. So werden z.B. Buchstaben aus Zahlenfeldern automatisch entfernt. Um eingegebene Daten in die XML-Datei zu übernehmen, muss der *Änderungen Übernehmen*-Knopf gedrückt werden.
Der *Neues Element*-Knopf öffnet ein Auswahlfenster, in dem man den jeweiligen neu zu erstellenden Elementtyp auswählt.
Der *Element löschen*-Knopf löscht das aktuell ausgewählte Element.

Literaturverzeichnis

[Harold 05]
Harold, Elliotte Rusty: Means, W. Scott: XML in a Nutshell. – 3.Aufl. – Köln: O'Reilly, 2005

[Krüger 03]
Krüger, Guido: Handbuch der Java-Programmierung. – 3.Aufl. – München: Addison Wesley Verlag, 2003

[Laughlin 06]
McLaughlin, Brett D.: Edelson, Justin: Java & XML. – 3.Aufl. – Sebastopol: O'Reilly, 2006

[Stahl 07]
Stahl, Thomas: Völter, Markus: Efftinge, Sven: Haase, Arno: Modellgetriebene Software-Entwicklung. – 2.Aufl. – Heidelberg: dpunkt.verlag GmbH, 2007

[Bourette]
Borette, Ronald: XML and Databases. http://www.rpbourret.com/, verfügbar am 25.3.2010

[JavaTutorial]
Oracle Corporation: The Java Tutorial. http://java.sun.com/docs/books/tutorial/, verfügbar am 25.3.2010.

[JAXFront]
xcentric technology & consulting GmbH: JAXFront. http://www.jaxfront.org/, verfügbar am 25.3.2010.

[JCalendar]
Tödter, Kai: JCalendar. http://www.toedter.com/, verfügbar am 25.3.2010.

[XAmple]
Golubov, Felix: XAmple. http://www.felixgolubov.com/XMLEditor/, verfügbar am 25.3.2010.

[Xerces]
The Apache Software Foundation: Apache Xerces. http://xerces.apache.org/, verfügbar am 25.3.2010.

[Boensch 09]
Bönsch, Steffen: Untersuchungen zur Software-Produktlinien-Erstellung für eine Enterprise-Applikation unter Verwendung der modellgetriebenen Software-Architektur (MDA) und der modellgetriebenen Software-Entwicklung (MDSD). – 2009. - 80 S.
Hochschule Mittweida, Fakultät Mathematik/Naturwissenschaften/Informatik, Diplomarbeit, 2009

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegeben Literatur und Hilfsmittel angefertigt habe.

Mittweida, der …… Unterschrift ……