Egg

Ein Generator-Generator für Quellkode-Stylesheets

Alexander Knecht

Version: 0.4

28. Februar 2004

Historie

Version	Status	Datum	Autor(en)	Erläuterung
0.1	in Arbeit	01.06.01	A. Knecht	Erster Wurf
0.2	in Arbeit	26.07.01	A. Knecht	Kapitel ‚Installation' hinzugefügt.
0.3	quasi fertig	13.01.02	A. Knecht	Kosmetik.
0.4	fertig	29.02.04	A. Knecht	Abschnitte 'Weitere Anweisungen', 'Mustermengen' und 'Optionen' hinzugefügt.

Inhaltsverzeichnis

HISTORIE III

1 MOTIVATION 1

2 GENERATOR-ENTWICKLUNGSPROZESS 2

3 GENERATOR-DEFINITION 8

3.1 PFADAUSDRÜCKE 9
3.2 ANWEISUNGEN 11
3.2.1 Musterdefinition 11
3.2.2 Musteranwendung 12
3.2.3 Weitere Anweisungen 13
3.2.4 Mustermengen 16
3.3 OPTIONEN 19
3.3.1 Knotenvariable Optionen 21
3.3.2 Ausgabedatei Optionen 23
3.3.3 Whitespace Optionen 25
3.4 BUILTIN-METHODEN 27

4 GENERATOR-AUFRUF 28

5 EGG-AUFRUF 28

6 EGG-INSTALLATION 28

REFERENZEN 30

1 Motivation

In diesem Dokument wird das Werkzeug *Egg* zur Erstellung von Generatoren beschrieben. Diese Generatoren erhalten jeweils eine XML-Datei als Eingabe und schreiben ihre Ausgabe in Text-Dateien. Die Transformation der XML-Inhalte kann in Java frei ausprogrammiert werden. Neben Java reichen Grundkenntnisse in XML, und regulären Ausdrücken, um Generator-Definitionen zu erstellen. Ein Verständnis der XSL-Idee ist hilfreich. Siehe dazu auch [1] und [2].

Die Generator-Definition entspricht vom Konzept her einem XSL-Stylesheet. Über Pfadausdrücke werden Knoten in der XML-Eingabedatei referenziert und deren Inhalte gelesen. Diese Daten können dann nach Wunsch manipuliert und als Ergebnis ausgegeben werden. Damit drängt sich folgende Frage auf:

Wozu ist ein weiteres Werkzeug nötig, wenn es doch XSL gibt?

1. Beschränktes Programmiermodell

   XSL hat eine eigene Programmiersprache zur Beschreibung von Transformationen. Komplexere Funktionen werden wegen der Einbettung der Befehle in XML-Tags schnell unübersichtlich. Die standardmäßigen Basisfunktionen bieten nur einen beschränkten Leistungsumfang. Weiterhin ist es nicht möglich Unterprogramme zu definieren bzw. Bibliotheksfunktionen zu importieren.

   Egg-Ansatz: Die Stylesheet-Funktionalität (Matchen von Pfadausdrücken in XML-Dateien, Steuern der Ausgabe) wird in eine bekannte, verbreitete Programmiersprache eingebettet. Somit kann man alle Features der Programmiersprache und -umgebung nutzen. Der Lernaufwand wird reduziert.

   Im vorliegenden Werkzeug erfolgte die Einbettung in Java. Andere Sprachen wären ebenfalls denkbar (‚Egg for Perl‘, ‚Egg for Tcl‘, ...).

2. Keine Whitespace-Kontrolle

   XSL bietet keine ausreichende Kontrolle über Leerzeichen im Ausgabestrom. Da der Fokus auf der Transformation von XML-Bäumen liegt wurde auf die Steuerung der Whitespaces in der Ausgabe offenbar keinen größeren Wert gelegt.

   Egg ist insbesondere zum Bau von Kode-Generatoren gedacht. Hierbei ist es wichtig Zeilenumbrüche, Einrückungen und Tabulatoren gezielt einsetzen zu können.

   Egg-Ansatz: Generell sollte generierter Quellkode lesbar sein und so aussehen, als sei er manuell geschrieben worden. D.h. die optische Darstellung mit Leerzeichen muss auch in der Generator-Ausgabe enthalten sein und daher in der Generator-Definition spezifiziert werden können.

3. Generator-Ausgabe ist nicht direkt als Generator-Definition verwendbar

   XSL-Stylesheets sind in XML-Format beschrieben. D. h. die Generator-Ausgaben müssen im XSL-Stylesheet auch durch den XML-Parser des XSL-Transformators. Dazu müssen normale Textpassagen der Generator-Ausgabe in

CDATA-Sektionen gestellt werden so dass der Bezug von Generator-Ausgabe und Darstellung in der Generator-Definition nicht direkt gegeben ist.

Egg-Ansatz: Im Entwicklungsprozess von (Kode-)Generatoren wird zuerst der später zu generierende Kode manuell erstellt und in der Zielumgebung getestet. Dies stellt den Prototyp der Generator-Ausgabe dar, in dem dann Muster identifiziert und parametriert werden, um zur Generator-Definition zu gelangen.

Das Ziel ist den schon lauffähigen Prototyp-Kode dabei sowenig wie möglich zu verändern, um zum einen den Generatorbau zu beschleunigen und zum anderen Fehlerquellen bei Transformation in die Generator-Definition zu reduzieren.

2 Generator-Entwicklungsprozess

Die Triebfeder zum Bau von Generatoren ist wie so oft der Wunsch Inhalt und Form zu trennen. Im Kontext von Kode-Generierung geht es darum Fachliches, Anwendungsspezifisches von technischen Entwurfsentscheidungen zu entkoppeln und separate Pflege zu ermöglichen.

Vorgehen

Zur Entwicklung eines Generators hat sich die folgende Vorgehensweise bewährt. Diese vier Schritte der Generator-Definition werden im danach anhand eines Beispiels erläutert.

1. *Prototyp-Erstellung*: Zuerst wird ein Prototyp der Generator-Ausgabe erstellt und in der Zielumgebung getestet.
2. *AT-Analyse*: Dabei werden im Ausgabe-Prototyp die anwendungsspezifischen Begriffe (A) identifiziert und somit von den rein technisch bedingten Textbereichen (T) getrennt.
3. *Musterdefinition*: Hierbei werden die A- und T-Teile in zwei Dateien ausgelagert und Redundanzen zusammengefasst. Die A-Stellen werden im T-Teil parametriert und in einem XML-Dokument zusammengefasst. Die Parameternamen aus dem T-Teil tauchen als Attributnamen im XML-Dokument wieder auf, um den Bezug zum A-Teil herzustellen.
4. *Generator-Definition*: Dazu werden die Musterdefinitionen über Pfadausdrücke mit den gewünschten XML-Knotenmengen verknüpft und die Reihenfolge der Musteranwendung bestimmt. Des weitern wird ggf. benötigte Transformationslogik von XML-Inhalten in Generator-Ausgaben hinzugefügt und die Ausgabedateien festgelegt.

Beispiel

Im Rahmen einer Web-Anwendung wird pro Dialog eine Java-Klasse benötigt, die die Brücke von Benutzeroberfläche und Anwendungskern schlägt. In diesem so genannten Dialogmodell gibt es pro Dialogfeld zwei Attribute. Das eine repräsentiert das Dialogelement auf der Oberfläche (Widget) und das andere speichert den Wert, den das Datenfeld aus Sicht des Anwendungskerns enthält. Die Dialogmodell-Klasse enthält dann noch einen Konstruktor, der die Attribute initialisiert sowie Getter, um auf die Dialogfelder zuzugreifen und Getter/Setter für die Datenfelder.

In dem Beispiel wird nun eine Generator-Definition erstellt, um die Dialogmodell-Klassen zu erzeugen.

Prototyp-Erstellung

In Abbildung 1 ist der Prototyp der Generator-Ausgabe für den Dialog ‚Benutzerverwaltung' dargestellt. Über den ‚Benutzerverwaltung'-Dialog können die Kontaktdaten eines Benutzers gepflegt werden. Er umfasst die Dialogfelder ‚Anrede', ‚Nachname', ‚Vorname', ‚Straße', ‚PLZ', ‚Ort', ‚Telefon', ‚Telefax', ‚Email' und ‚Organisation'. Für die Erstellung des Dialogmodell-Prototypen werden nur exemplarisch die Dialogfelder ‚Anrede' und ‚Nachname' implementiert.

AT-Analyse

Abbildung 2 zeigt den Ausgabe-Prototyp nach der AT-Analyse. Alle fachlichen Begriffe sind markiert. Der T-Teil bleibt unverändert.

In Abbildung 3 werden nun die A-Begriffe durch Parameter ersetzt und die dabei entstehenden Redundanzen eliminiert. Die Parameternamen erscheinen in spitzen Klammern.

Musterdefinition

In Abbildung 4 werden die ‚ausgeklammerten' A-Teile in einem XML-Dokument zusammengefasst. Die weiteren Dialogfelder, die im Prototypen fehlen, werden noch hinzugefügt.

Generator-Definition

In Abbildung 5 werden dann die Egg-Anweisungen in den parametrierten Prototyp aus Abbildung 3 eingefügt. Die ‚generator'- und ‚apply'-Anweisungen legen die Ausgabe-Muster fest. Die Syntax ist einer Java-Methodendefinition nachempfunden. In der Parameterliste der Musterdefinition wird per Pfadausdruck der XML-Knoten referenziert, der als Kontext für die Musteranwendung herangezogen wird. Die Pfadausdrücke beginnen mit einem Schrägstrich ‚/'.

In der ‚generator'-Anweisung wird die Knotenvariable ‚Dialog' mit dem XML-Knoten ‚Dialog' aus Abbildung 4 verknüpft. In den ‚apply'-Anweisungen werden allen XML-Knoten ‚gematcht', die direkt im aktuellen Kontext (‚Dialog'-Knoten') liegen. Dies die Dialogfelder ‚Optionsgruppe', ‚Eingabefeld' und ‚Auswahlfeld'.

Innerhalb der Musterdefinitionen treten Variablendefinitionen auf. In dem rechten Ausdruck der Zuweisung werden Zugriffe auf Egg-Variablen, XML-Elemente und -Attribute in spitze Klammern gesetzt. Die ‚File'-Variable in der zweiten Zeile legt die Ausgabe-Datei fest.

Um jetzt den Dialogmodell-Generator zu erstellen, speichert man die Generator-Definition unter ‚DialogmodellGenerator.egg' und ruft damit als Parameter das Egg-Werkzeug auf. Es entsteht u. A. eine ‚DialogmodellGenerator.bat'-Datei, die mit der XML-Datei als Parameter aufgerufen werden kann. Dies stößt die eigentliche Erzeugung der Generator-Ausgabe an, die dann in der ‚BenutzerverwaltungDM.java'-Datei erscheint.

```
import de.vkb.framework.datatype.*;    // Datenfeld-Typen
import de.vkb.framework.dialog.*;      // Dialogfeld-Typen

public class BenutzerverwaltungDM extends DialogModel {

    // Datenfelder
    private Anrede mAnrede;
    private Text mName;
```

```
    // Dialogfelder
    private Optionsgruppe mAnredeDF;
    private Eingabefeld mNameDF;

    // Konstruktor
    public BenutzerverwaltungDM() {

        // Datenfelder initialisieren
        setAnrede(Anrede.EMPTY_ANREDE);
        setName(Text.EMPTY_TEXT);

        // Dialogfelder initialisieren
        mAnredeDF = new Optionsgruppe("Anrede");
        mNameDF = new Eingabefeld("Nachname");
    }

    // Getter für Dialogfelder
    public Optionsgruppe getAnredeDF() {
        return mAnredeDF;
    }
    public Eingabefeld getNameDF() {
        return mNameDF;
    }

    // Getter/Setter für Datenfelder
    public Anrede getAnrede() {
        return mAnrede;
    }
    public void setAnrede(Anrede pAnrede) {
        mAnrede = pAnrede;
        mAnredeDF.setWert (pAnrede);
    }

    public Text getName() {
        return mName;
    }
    public void setName(Text pName) {
        mName = pName;
        mNameDF.setWert (pName);
    }
} // end-of Dialogmodel 'Benutzerverwaltung'
```

Abbildung 1: Prototyp der Generator-Ausgabe

```
import de.vkb.framework.datatype.*;    // Datenfeld-Typen
import de.vkb.framework.dialog.*;      // Dialogfeld-Typen

public class BenutzerverwaltungDM extends DialogModel {

    // Datenfelder
    private Anrede mAnrede;
```

```
    private Text mName;

    // Dialogfelder
    private Optionsgruppe mAnredeDF;
    private Eingabefeld mNameDF;

    // Konstruktor
    public BenutzerverwaltungDM() {

        // Datenfelder initialisieren
        setAnrede(Anrede.EMPTY_ANREDE);
        setName(Text.EMPTY_TEXT);

        // Dialogfelder initialisieren
        mAnredeDF = new Optionsgruppe("Anrede");
        mNameDF = new Eingabefeld("Nachname");
    }

    // Getter für Dialogfelder
    public Optionsgruppe getAnredeDF() {
        return mAnredeDF;
    }
    public Eingabefeld getNameDF() {
        return mNameDF;
    }

    // Getter/Setter für Datenfelder
    public Anrede getAnrede() {
        return mAnrede;
    }
    public void setAnrede(Anrede pAnrede) {
        mAnrede = pAnrede;
        mAnredeDF.setWert (pAnrede);
    }

    public Text getName() {
        return mName;
    }
    public void setName(Text pName) {
        mName = pName;
        mNameDF.setWert (pName);
    }
} // end-of Dialogmodel 'Benutzerverwaltung'
```

Abbildung 2: Prototyp der Generator-Ausgabe nach AT-Analyse

```
import de.vkb.framework.datatype.*;    // Datenfeld-Typen
import de.vkb.framework.dialog.*;      // Dialogfeld-Typen

public class <Dialog>DM extends DialogModel {

    // Datenfelder
```

```
    private <Datentyp> m<Feldname>;

    // Dialogfelder
    private <Feldtyp> m<Feldname>DF;

    // Konstruktor
    public <Dialog>DM() {

        // Datenfelder initialisieren
        set<Feldname>(<Datentyp>.EMPTY_<DATENTYP>);

        // Dialogfelder initialisieren
        m<Feldname>DF = new <Feldtyp>("<Beschriftung>");
}

    // Getter für Dialogfelder
    public <Feldtyp> get<Feldname>DF() {
        return m<Feldname>DF;
    }

    // Getter/Setter für Datenfelder
    public <Datentyp> get<Feldname>() {
        return m<Feldname>;
    }
    public void set<Feldname>(<Datentyp> p<Feldname>) {
        m<Feldname> = p<Feldname>;
        m<Feldname>DF.setWert (p<Feldname>);
    }
}  // end-of Dialogmodel '<Dialog>'
```

Abbildung 3: Parametrierte Prototyp-Ausgabe

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Dialog Name="Benutzerverwaltung">
  <!-- Dialogfelder aus Prototyp der Generator-Ausgabe -->
  <Optionsgruppe Name="Anrede" Datentyp="Anrede"/>
  <Eingabefeld Name="Name" Datentyp="Text" Beschriftung="Nachname"/>

  <!-- weitere Dialogfelder für die Generator-Ausgabe -->
  <Eingabefeld Name="Vorname" Datentyp="Text"/>
  <Eingabefeld Name="Strasse" Datentyp="Text" Beschriftung="Straße"/>
  <Eingabefeld Name="PLZ" Datentyp="Text"/>
  <Eingabefeld Name="Ort" Datentyp="Text"/>
  <Eingabefeld Name="Telefon" Datentyp="Telefon"/>
  <Eingabefeld Name="Telefax" Datentyp="Telefon"/>
  <Eingabefeld Name="EMail" Datentyp="EMail" Beschriftung="E-Mail"/>
  <Auswahlfeld Name="Organisation" Datentyp="Text"/>
</Dialog>
```

Abbildung 4: Anwendungsspezifischer Teil als XML-Dokument

```
.generator DialogmodellGenerator ( Node Dialog = </Dialog> ) {
```

```
	.File outputFile = <Dialog> + "DM.java";
	import de.vkb.framework.datatype.*;	// Datenfeld-Typen
	import de.vkb.framework.dialog.*;	// Dialogfeld-Typen

	public class <Dialog>DM extends DialogModel {

		// Datenfelder
	.apply DatenfeldDefinition ( Node Feld = </.> ) {
		.String Datentyp = </Datentyp>;
		private <Datentyp> m<Feld>;
	.}

		// Dialogfelder
	.apply DialogfeldDefinition ( Node Feld = </.> ) {
		.String Feldtyp = <Feld>.getDomNode().getNodeName();
		private <Feldtyp> m<Feld>DF;
	.}

		// Konstruktor
		public <Dialog>DM() {

			// Datenfelder initialisieren
		.apply DatenfeldInit ( Node Feld = </.> ) {
			.String Datentyp = </Datentyp>;
			.String DATENTYP = <Datentyp>.toUpperCase();
			set<Feld>(<Datentyp>.EMPTY_<DATENTYP>);
		.}

			// Dialogfelder initialisieren
		.apply DialogfeldInit ( Node Feld = </.> ) {
			.String Feldtyp =
<Feld>.getDomNode().getNodeName();
			.String Beschriftung = </Beschriftung>.equals("") ?
</Name> : </Beschriftung>;
			m<Feld>DF = new <Feldtyp>("<Beschriftung>");
		.}
		}

		// Getter für Dialogfelder
	.apply DialogfeldGetter ( Node Feld = </.> ) {
		.String Feldtyp = <Feld>.getDomNode().getNodeName();
		public <Feldtyp> get<Feld>DF() {
			return m<Feld>DF;
		}
	.}

		// Getter/Setter für Datenfelder
	.apply DatenfeldGetterSetter ( Node Feld = </.> ) {
		.String Datentyp = </Datentyp>;
		public <Datentyp> get<Feld>() {
			return m<Feld>;
		}
		public void set<Feld>(<Datentyp> p<Feld>) {
			m<Feld> = p<Feld>;
```

```
                    m<Feld>DF.setWert (p<Feld>);
                }

        .}
        }  // end-of Dialogmodel '<Dialog>'
.}
```

Abbildung 5: Vollständige Generator-Definition zum Prototypen

3 Generator-Definition

Eine Generator-Definition ist eine Folge von Anweisungen, die sequentiell abgearbeitet werden, wie man es von imperativen Programmiersprachen kennt. Es gibt verschiedene Anweisungen für die Steuerung des Generators und eine Anweisung, mit der die Ausgaben des Generators festgelegt werden.

Aus dem Generator-Entwicklungsprozess wird ersichtlich, dass zunächst die Generator-Ausgabe aus dem Ausgabe-Prototyp direkt übernommen wird und danach zusätzliche Zeilen als Anweisungen eingefügt werden, um die Generator-Struktur und -Funktionalität festzulegen.

Das Ziel ist hierbei die Generator-Definition so nahe wie möglich an der Generator-Ausgabe zu halten, um 'mit einem Blick' zu erfassen, wie die Ausgabe erscheint. Wie gut sich dieses Ziel erreichen lässt hängt stark vom Verhältnis der Anzahl der Ausgabe-Anweisungen zu der Anzahl der zusätzlichen Steuerungsanweisungen. Wenn nur sehr wenige Steuerungsanweisungen nötig sind, tritt die Generator-Ausgabe klarer hervor. Die Generator-Definition sieht dann mehr nach einem Template aus. Umgekehrt, wenn viel Transformationslogik nötig ist und ehr wenige Ausgabe-Anweisungen gebraucht werden, siehe die Generator--Definition mehr nach einem imperativen Programm aus. Das Verhältnis von Generator-Ausgabe und -Logik ist vom jeweiligen Anwendungsfall abhängig.

Zeilenmarker

Um die Ausgabe-Anweisungen eins zu eins aus dem Generator-Prototyp übernehmen zu können, ist die Generator-Definition zeilenweise aufgebaut. Für die Ausgabe-Zeilen kann keine bestimmte Struktur angenommen werden, da sie vom konkreten Anwendungsfall abhängig sind. Um die Generator-Anweisungen dennoch von den Ausgabe-Zeilen zu unterscheiden, werden sie mit einer speziellen Markierung, dem Zeilenmarker, versehen.

Der *Zeilenmarker* ist ein einzelnes Zeichen, mit dem alle Egg-Anweisungszeilen beginnen müssen, die nicht der Ausgabe dienen. Whitespaces am Zeilenanfang sind hierbei zulässig. Die ‚generator'-Anweisung legt den Zeilenmarker fest, d. h. das Zeichen vor dem ‚generator'-Schlüsselwort ist der Zeilenmarker für die gesamte Generator-Definition. In Abbildung 5 wird beispielsweise der Punkt ‚.' verwendet.

Aufgrund des zeilenbasierten Aufbaus einer Generator-Definition können lange Anweisungen nicht in mehrere Zeilen umgebrochen werden, wie man es vielleicht von anderen Programmiersprachen her kennt.

Einzug

Auf der einen Seite möchte man eine Generator-Definition wie gewohnt als Folge von Anweisung aufschreiben und die innere Struktur durch Einrückung mit Tabulatoren sichtbar machen. Durch die direkte Übernahme der Ausgabe-Zeilen als Egg-Anweisungen, kann es schnell unübersichtlich werden, wenn in der Ausgabe ebenfalls Einzüge vorkommen, was gerade bei der Erzeugung von Quellkode häufig der Fall ist.

Um den Blick für den Kontrollfluss der Generator-Definition nicht völlig zu verlieren, können die Einzüge der Ausgabe-Zeilen denen der Steueranweisungen angepasst werden, so dass die Schachtelung der Steueranweisungen sichtbar bleibt. Die Generator-Definition wird so lesbarer, jedoch muss man nun bei den Tabulatoren der Ausgabe-Anweisungen unterscheiden, ob sie nur zur Übersichtlichkeit in der Eingabe dienen, oder ob sie auch in die Ausgabe gehören.

Zur Unterscheidung der Eingabe- und Ausgabe-Einzüge wird ein *linker Rand* für eine Anweisung eingeführt. Er ergibt sich einfach durch die Anzahl der Tabulatoren links vom Zeilenmarker der Anweisung. Für Ausgabe-Zeilen, die innerhalb der Anweisung geschachtelt sind, werden dann alle Tabulatoren links von diesem Rand entfernt und nur die Tabulatoren rechts davon in die Ausgabe übernommen (siehe dazu auch 'indent'-Option).

3.1 Pfadausdrücke

Pfadausdrücke dienen zur Referenzierung von Knoten im XML-Eingabedokument. Ihre Syntax ist an die der regulären Ausdrücke angelehnt, die zur Mustererkennung in Zeichenketten verwendet wird.

Die Egg-Pfadausdrücke sind einfach gehalten, um ihre Verwendung nicht unnötig zu verkomplizieren und ihre Verarbeitung im Egg-Werkzeug zu erleichtern. Es hat sich allerdings herausgestellt, dass komplexere Pfadausdrücke in manchen Fällen durchaus nützlich sein können. Deshalb ist es sinnvoll die Egg-Pfadausdrücke später einmal durch XPath-Ausdrücke zu ersetzen (siehe [3]).

Syntax

Die folgende Abbildung 6 gibt die Syntax der Egg-Pfadausdrücke in der Backus-Naur-Form an:

```
             path-expr := "/"
                        | path-expr-steps

       path-expr-steps := "/" path-expr-step
                        | "/" path-expr-step path-expr-steps

        path-expr-step := [not-operator] node-name [axis-specifier]

          not-operator := "^"
             node-name := <XML-Nodename> | "#text" | "."
        axis-specifier := child-or-self-axis | descedant-or-self-
axis
     child-or-self-axis := "?"
```

```
descendant-or-self-axis := "*"
```

Abbildung 6: BNF für Egg-Pfadausdrücke

Semantik

Ein *Name eines Knotens* ist entweder der Name eines XML-Elements oder der eines XML-Attributs. Der XML-Knotentyp wird also nicht unterschieden. Ein *Pfad eines Knotens* ist der Weg im Knotenbaum von einem Kontextknoten (exklusive) bis zum Knoten selbst (inklusive). Beispielsweise sind die Knotenpfade zu den Dialogfeldern ‚Anrede' und ‚Nachname' aus Abbildung 4 relativ zur Wurzel im Dokumentenbaum gesehen:

- `/Dialog`
- `/Dialog/Optionsgruppe`
- `/Dialog/Optionsgruppe/Name`
- `/Dialog/Optionsgruppe/Datentyp`
- `/Dialog/Eingabefeld`
- `/Dialog/Eingabefeld/Name`
- `/Dialog/Eingabefeld/Datentyp`
- `/Dialog/Eingabefeld/Beschriftung`

Der *Wert eines Knotens* ist bei XML-Attributen der Wert des Attributs. Bei XML-Elementen wird der Knotenwert aus einem bestimmten Attribut gelesen. Standardmäßig ist dies das ‚Name'-Attribut. Das Standardattribut sowie die Wertbestimmung kann durch Optionen noch genauer beeinflusst werden. Siehe Abschnitt 3.3.1.

Ein *Pfadausdruck* ist nun ein Muster, um eine Menge von Knotenpfaden zu beschreiben. Er besteht aus Einzelschritten, die als Testbedingung der Kontennamen betrachtet werden können. Ein *Knotenpfad passt auf einen Pfadausdruck*, wenn alle Testbedingungen der Einzelschritte erfüllt sind. Die *Knotenliste zu einem Pfadausdruck* enthält alle Knoten, deren Knotenpfade relativ zu einem Kontextknoten auf den Pfadausdruck passen.

Zu diesem Mustertest geht man den Pfadausdruck schrittweise durch und vergleicht die Einzelschritte mit dem Knotenpfad. Sind die Knotennamen gleich, dann ist die Testbedingung für den Einzelschritt erfüllt und man kann den nächsten Schritt im Pfadausdruck prüfen.

Der Test der Einzelschritte kann durch Modifikatoren beeinflusst werden. Mit dem Not-Operator ‚^' ist die Bedingung erfüllt, wenn die Knotennamen ungleich sind. Mit dem ‚?'-Operator wird die Bedingung optional, d. h. der Einzelschritt der Pfadausdrucks kann im Knotenpfad stehen - muss aber nicht. Mit dem ‚*'-Operator kann der Einzelschritt beliebig oft im Knotenpfad vorkommen - oder auch gar nicht.

Des Weiteren kann als Knotenname auch eine *Alternativenliste* angegeben werden. Die einzelnen Ausprägungen werden in runde Klammern eingeschlossen und durch ‚|' getrennt. Die Testbedingung ist erfüllt, wenn ein Knotenname aus der Liste passt. Schließlich kann als Knotenname auch ein Punkt ‚.' als *Wildcard* angegeben werden. Er steht dann für beliebige Knotennamen, jedoch nicht für Textknoten (#text).

Beisiele

- `/Dialog/./Datentyp`

  Die Knotenliste enthält alle Datentyp-Attribute aus Abbildung 4.
- `/^Liste*/(Eingabefeld|Ausgabefeld)`

Die Kontenliste enthält alle Ein- und Ausgabefeld-Knoten im aktuellen Kontext, in deren Knotenpfad keine ‚Liste' vorkommt.

- `/.*/Liste/.*/(Eingabefeld|Ausgabefeld)`

  Findet alle Ein- und Ausgabefelder innerhalb einer Liste.

3.2 Anweisungen

Die Reihenfolge der Musteranwendungen sowie die Transformationslogik folgen dem imperativen Programmierparadigma. Eine Generator-Definition ist demnach eine Folge von Anweisungen, die in einem Skript werden nacheinander abgearbeitet. Abbildung 7 gibt die BNF für *Muster-Skripte* und die möglichen Anweisungen an, auf die in den folgenden Abschnitten näher eingegangen wird.

```
    pattern-script := [pattern-statements]

pattern-statements := pattern-statement
                    | pattern-statement pattern-statements

 pattern-statement := stmt-apply-pattern
                    | stmt-define-body
                    | stmt-define-head
                    | stmt-define-pattern
                    | stmt-define-variable
                    | stmt-define-version
                    | stmt-if-then-else
                    | stmt-match-pattern
                    | stmt-match-patternset
                    | stmt-write-output
```

Abbildung 7: BNF für Muster-Skripte und mögliche Anweisungen

3.2.1 Musterdefinition

Eine *Musterdefinition* (*pattern*) ist direkt mit einer Methoden- oder Prozedur-Definition vergleichbar. Sie erhalten einen Namen, über den sie referenziert werden können, sowie eine Liste mit formalen Parametern, die bei der Musteranwendung übergeben werden und schließlich ein Skript, das die auszuführenden Anweisungen enthält.

In Abbildung 8 ist die Syntax dargestellt, die direkt von Java abgeleitet ist. Ebenso entsprechen die aufgeführten Egg-Typen den jeweiligen Java-Typen: `java.lang.String`, `java.io.File`, `org.w3c.dom.Node`, `java.lang.Object` und den Basistypen `boolean` und `int`.

```
  stmt-define-pattern := "pattern" p-name formal-parameters "{"
                                   pattern-script
```

```
                          "}"

               p-name := <java identifier>
        variable-name := <java identifier>
             egg-type := "String" | "File" | "Node" | "Object"
                       | "boolean" | "int"

    formal-parameters := "(" [formal-parameter-list] ")"
formal-parameter-list := formal-parameter
                       | formal-parameter "," formal-parameter-list
     formal-parameter := egg-type variable-name
```

Abbildung 8: BNF zur Musterdefinition

3.2.2 Musteranwendung

Die *Musteranwendung* ist das Gegenstück zur Musterdefinition, wie etwa der Methoden-Aufruf zur Methoden-Definition. Es gibt sie in zwei Ausprägungen, wie in Abbildung 9 ersichtlich.

In der ersten Form wird eine vorangegangene Musterdefinition mit dem entsprechenden Muster-Namen referenziert. Die Übergabe eventueller Parameter erfolgt wie bei einem Java-Methodenaufruf.

In der zweiten Form erfolgen die Anwendung und die Definition zusammen an derselben Stelle (*Inline-Definition*). Eventuelle konkrete Parameter werden wie bei einer Variabeleninitialisierung direkt den formalen Parametern zugewiesen. Dabei müssen die Typen natürlich übereinstimmen.

Bei der Übergabe eines Pfadausdrucks kann der Parametertyp entweder `String` oder `Node` sein. Für `String`-Parameter wird der Wert des ersten Knotens aus der Kontenliste zum Pfadausdruck verwendet. Für `Node`-Parameter wird das Muster für jeden Knoten aus der Knotenliste zum Pfadausdruck einmal angewendet und dabei der jeweilige Knotenwert übergeben. Diese Schleifenbehandlung erfolgt nur für die *Knotenvariable* des Musters, d. h. für den ersten `Node`-Parameter in der Parameterliste des Musters. Alle weitern werden als normale Parameter übergeben.

Bei der Abarbeitung eines Muster-Skriptes gibt es einen *Kontextknoten* in der XML-Eingabedatei. Alle Pfadausdrücke innerhalb des Musters werden relativ zu diesem Knoten ausgewertet. Bei der Anwendung eines Musters wird dann der, in der Knotenvariable übergebene, Knoten zum Kontextknoten des aufgerufenen Musters.

Beispielsweise ist in Abbildung 5 der Kontextknoten im Muster ‚DialogmodellGenerator' der ‚Dialog'-Parameter. Im Muster ‚DatenfeldDefinition' ist es der ‚Feld'-Parameter.

```
     stmt-apply-pattern := "apply" p-name concrete-parameters ";"

                         | "apply" p-name inline-parameters "{"
                               pattern-script
                           "}"
```

```
    concrete-parameters := "(" [concrete-parameter-list] ")"
concrete-parameter-list := concrete-parameter
                         | concrete-parameter "," concrete-parameter-
list
     concrete-parameter := egg-expr

      inline-parameters := "(" [inline-parameter-list] ")"
  inline-parameter-list := inline-parameter
                         | inline-parameter "," inline-parameter-list
       inline-parameter := egg-type variable-name "=" egg-expr

               egg-expr := "<" (variable-name | path-expr) ">"

  stmt-define-generator := "generator" p-name inline-parameters "{"
                              pattern-script
                           "}"
```

Abbildung 9. BNF zur Musteranwendung

Die *Generator-Anweisung* (stmt-define-generator) aus Abbildung 9 ist der Einstiegspunkt zum Bau eines Generators. Sie muss als erste Anweisung in einer Egg-Datei stehen. Alle weiteren Anweisungen stehen dann in ihrem Muster-Skript.

Die Inline-Parameterliste der Generator-Anweisung enthält nur eine Knotenvariable. Der Kontextknoten für den Pfadausdruck des `Node`-Parameters ist die Dokumentenwurzel der XML-Eingabedatei.

3.2.3 Weitere Anweisungen

Zusätzlich zur reinen Musterdefinition und -anwendung gibt es in Abbildung 10 weitere Anweisungen zur Beschreibung eines Generators:

```
stmt-define-variable := egg-type variable-name "=" assignment-expr
";"

     assignment-expr := (<java expr code> | egg-expr)*
       variable-expr := "<" variable-name ">"

   stmt-write-output := (<output-text> | variable-expr)*

   stmt-if-then-else := "if" "(" variable-expr ")" "{"
                           pattern-script
                        "}" ["else" "{"
                           pattern-script
                        "}"]

 stmt-define-version := <version string>

    stmt-define-head := "head" "{"
                          <java class header>
```

```
                         "}"

    stmt-define-body := "body" "{"
                            <java class body>
                         "}"
```

Abbildung 10: BNF für weitere Egg-Anweisungen

Die *Variable-Anweisung* (stmt-define-variable) definiert und initialisiert eine lokale Variable im aktuellen Muster-Skript. Das Prinzip und die Syntax verhalten sich analog zu einer Variablen-Definition in Java. Abbildung 11 gibt einige Beispiele:

```
.boolean outDirGiven = getArgs().length > 0;
.String  outDir      = <outDirGiven> ? getArgs()[0] : "output";
.File    outFile     = <outDir> + File.separator + "result.txt";
.String  now = (new Date()).toString();
.
These are the latest Results (date: <now>) ...
```

Abbildung 11: Beispiele zur Variable-Anweisung

Der ersten `File`-Variablen innerhalb eines Muster-Skript kommt dabei eine besondere Bedeutung zu. Sie legt die Datei fest, in die die *aktuelle Ausgabe* geschrieben wird. Im Beispiel in Abbildung 11 würden weitere Generator-Ausgaben in die Datei 'result.txt' geschrieben. Die Ausgabedatei gilt für das aktuelle Muster-Skript sowie alle die darin aufgerufenen oder geschachtelten Muster. Während der Generator-Ausführung gibt es immer nur eine aktuelle Ausgabedatei. Es können aber mehrere Dateien nacheinander geschrieben werden.

Die *Ausgabe-Anweisung* (stmt-write-output) besteht aus einer Zeile Text, die nicht mit dem Egg-Zeilenmarker beginnt. In der Textzeile können Variablen-Ausdrücke (variable-expr) vorkommen. Diese Zeile wird unverändert an die aktuelle Ausgabe angehängt. Die Variablen werden dabei mit ihrem aktuellen Wert ausgegeben. Wird eine Knotenvariable im Ausgabetext verwendet, so wird ihr Knotenwert ausgegeben. Ein Beispiel ist die letzte Zeile in Abbildung 11.

Die *If-Anweisung* (stmt-if-then-else) erlaubt eine Verzweigung beim Abarbeiten des aktuellen Muster-Skripts. Die Anweisung verhält sich so, wie man es von gängigen Programmiersprachen gewohnt ist. Allerdings kann als Verzweigungsbedingung nur eine `boolean`-Variable verwendet werden. Die Angabe eines Ausdrucks ist derzeit nicht möglich.

Abbildung 12 zeigt die Verwendung der If-Anweisung am Beispiel eines Generators zur Erzeugung eines SQL-Skripts zum Importieren eines Typsystems.

```
.generator SqlTypeLoader ( </TypeDefinition> ) {
    .apply CreateType ( Node pType = </Type> ) [indent=-1] {
        .String lExtends      = </extends>;
        .boolean lHasBaseType = !lExtends.equals("");
```

```
        PROMPT Loading Type '<pType>' ...
        .if (<lHasBaseType>) {
            -- create type '<pType>' with base-type '<lExtends>'.
            exec TYPE_UTIL.ExtendType('<pType>', '<lExtends>');
        .} else {
            -- create type '<pType>'.
            exec TYPE_UTIL.CreateType('<pType>');
        .}
    .} // CreateType
.} // SqlTypeLoader
```

Abbildung 12: Beispiel zur If-Anweisung

Die *Version-Anweisung* (stmt-define-version) legt die Version des erzeugten Generators fest. Sie wird als Kontroll-Information in der Laufzeit-Ausgabe des Generators angezeigt und kann ein beliebiger Text sein. Im Bespiel in Abbildung 42 ist die Version '0.0.1' was auch der Voreinstellung entspricht.

Die *Head-Anweisung* (stmt-define-head) legt Java-Kode fest, der in den Kopfbereich der erzeugten Generator-Klasse kopiert wird. Hier können import-Anweisungen stehen, um externe Pakete zu verwenden. Standardmäßig sind die Pakete `java.lang.*`, `java.io.*` und `java.util.*` importiert.

```
.generator StmtHeadDemo ( </DateList> ) {
    .File lFile = "gen/full-dates.txt";
    .head {
        import java.text.DateFormat;
    .}
    .
    .apply DateTrafo ( Node pDate = </Date> ) {
        .Object lDate = DateFormat.getDateInstance().
                                          parse(<pDate>.toString());
        .String lFull = DateFormat.getDateInstance(DateFormat.FULL).
                                          format((Date)<lDate>);
        .
        full date of '<pDate>': '<lFull>'
    .}
.} // StmtHeadDemo
```

Abbildung 13: Beispiel zur Head-Anweisung

Die *Body-Anweisung* (stmt-define-body) legt Java-Kode fest, der in den Rumpf der erzeugten Generator-Klasse kopiert wird. Hier können Methoden-, Variablen- oder innere Klassen-Definitionen stehen, d. h. alles was in einer Java-Klasse verwendet werden kann. Die Body-Anweisungen können an beliebigen Stellen in der Generator-Definition vorkommen und ermöglichen freies Ausprogrammieren einer zusätzlich benötigten Transformationslogik.

In Abbildung 14 ist die Verwendung der Body-Anweisung anhand eines Generators zur Erzeugung von Java-Beans dargestellt:

```
.generator StmtBodyDemo ( Node pBean = </bean-def/bean> ) [fold-
key="name"] {
    .String lBean = firstUp(<pBean>.toString()) + "Bean";
    .File lFile = "gen/" + <lBean> + ".java";
    .
    .body {
        String firstUp ( String pName ) {
            if (pName.equals("")) {
                return "";
            }
            String lName = pName.substring(0,1).toUpperCase()
                         + pName.substring(1);
            return lName;
        }

        String getJavaType ( String pType ) {
            if ( pType.equals("number") ) {
                return "int";
            }
            if ( pType.equals("yesno") ) {
                return "boolean";
            }
            return "String";
        }
    .}
    .
    class <lBean> {
    .apply Property ( Node pProp = </property> ) {
        .String lProp = firstUp(<pProp>.toString());
        .String lType = getJavaType(</type>);

        // property '<pProp>'
        private <lType> m<lProp>;
        public <lType> get<lProp>() {
            return m<lProp>;
        }
        public void set<lProp>(<lType> p<lProp>) {
            m<lProp> = p<lProp>;
        }
    .}
    } // endof class '<lBean>'
.} // StmtBodyDemo
```

Abbildung 14: Beispiel zur Body-Anweisung

3.2.4 Mustermengen

Eine *Mustermenge* (*patternset*) ist eine Menge von Musterdefinitionen, die alternativ je nach Eingabe-Knoten angewendet werden. Die Reihenfolge der Musteranwendung wird durch die Reihenfolge der Knoten im Eingabedokument bestimmt. Im Prinzip

wird damit so etwas wie eine 'switch'-Anweisung über den aktuellen Eingabe-Knoten realisiert.

In Abbildung 15 sind die zusätzlichen Anweisungen in Mustermengen-Skripten aufgeführt.

```
    patternset-script := [patternset-statements]

patternset-statements := patternset-statement
                       | patternset-statement patternset-statements

 patternset-statement := pattern-statement
                       | stmt-define-patternset
                       | stmt-apply-patternset
                       | stmt-match-patternset
                       | stmt-match-pattern
```

Abbildung 15: BNF für Mustermengen-Skripte und mögliche Anweisungen

In Abbildung 16 sind die Anweisungen zur Definition und Anwendung von Mustermengen aufgeführt. Sie folgen der bekannten Form der 'stmt-define-pattern'- und 'stmt-apply-pattern'-Regeln.

```
stmt-define-patternset := "patternset" p-name formal-parameters "{"
                               patternset-script
                          "}"

 stmt-apply-patternset := "applyset" p-name concrete-parameters ";"
                        | "applyset" p-name inline-parameters "{"
                               patternset-script
                          "}"

 stmt-match-patternset := "matchset" p-name inline-parameters "{"
                              patternset-script
                          "}"

    stmt-match-pattern := "match" p-name inline-parameters "{"
                              pattern-script
                          "}"
```

Abbildung 16: BNF für Mustermengen-Anweisungen

Die folgenden Abbildungen zeigen die Verwendung von Mustermengen am Beispiel der Transformation einer Symbolleisten-Definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Toolbar Name="MyToolbar">
      <Button     Name="bu" />
      <Input      Name="in" />
      <Pushbutton Name="pb" />
```

```
      <Checkbox   Name="cb" />
</Toolbar>
```

Abbildung 17: XML-Eingabe für Mustermengen-Beispiel

In der nachfolgenden Abbildung 18 ist das Egg-Skript zur Transformation einer Symbolleiten-Definition angegeben. Es entspricht einer geschachtelten Fallunterscheidung. Ihre Ausführung beginnt mit der 'Toolbar'-Mustermenge. Ihre Knotenvariable wird mit einem Pfadausdruck initialisiert. Im Beispiel ist dies der aktuellen Kontext '/', der dem aktuellen 'Toolbar' Eingabeknoten entspricht. Er bildet den Kontext, in dem die einzelnen 'match'-Fälle in der Reihenfolge der Eingabe gesucht werden. Im Beispiel entspricht dies dem 'Button'-Element 'bu', dem 'Input'-Element 'in', dem 'Pushbutton'-Element 'pu' und schließlich dem 'Checkbox'-Element 'cb'. Für jedes dieser Eingabeknoten wird die passende 'match'-Anweisung ermittelt und das dazugehörige Muster-Skript abgearbeitet.

Vor und nach den 'match'-Anweisungen können Muster-Skripte stehen. Das *Prefix-Skript* wird vor jedem 'Match' ausgeführt. Das *Postfix-Skript* entsprechend nach jedem 'Match'. Dies gilt auch für geschachtelte Mustermengen. Innerhalb dieser Skrpte können neben Ausgabe- auch Variablen-Anweisungen oder Musteranwendungen vorkommen.

```
.generator PatternSetDemo ( Node pToolbar = </Toolbar> ) {
      .File lFile = "gen/toolbar-output.txt";
      .
      output for toolbar '<pToolbar>':
      -------------------------------------------------
      .applyset Toolbar (Node pItem = </>) {

            - prefix-script before a 'match' (<pItem>)
            .match Button (Node pButton = </Button>) {
                  -- script for 'Button' match (<pButton>)
            .}
            .match Pushbutton (Node pPushbutton = </Pushbutton>) {
                  -- script for 'Pushbutton' match (<pPushbutton>)
            .}
            .matchset Formfield() {
                  -- prefix-script for Formfield 'matchset'
                  .match Input (Node pInput = </Input>) {
                        --- script for 'Input' formfield (<pInput>)
                  .}
                  .match Checkbox (Node pCb = </Checkbox>) {
                        --- script for 'Checkbox' formfield (<pCb>)
                  .}
                  -- postfix-script for Formfield 'matchset'
            .}
            - postfix-script after a 'match' (<pItem>)
      .}
.} // PatternSetDemo
```

Abbildung 18: Egg-Skript für Mustermengen-Beispiel

Die Ausgabe nach Ausführung des 'PatternSetDemo'-Generators steht in der Datei 'gen/toolbar-output.txt' und ist in Abbildung 19 dargestellt:

```
output for toolbar 'MyToolbar':
--------------------------------------------------

      - prefix-script before a 'match' (bu)
      -- script for 'Button' match (bu)
      - postfix-script after a 'match' (bu)

      - prefix-script before a 'match' (in)
      -- prefix-script for Formfield 'matchset'
      --- script for 'Input' formfield (in)
      -- postfix-script for Formfield 'matchset'
      - postfix-script after a 'match' (in)

      - prefix-script before a 'match' (pb)
      -- script for 'Pushbutton' match (pb)
      - postfix-script after a 'match' (pb)

      - prefix-script before a 'match' (cb)
      -- prefix-script for Formfield 'matchset'
      --- script for 'Checkbox' formfield (cb)
      -- postfix-script for Formfield 'matchset'
      - postfix-script after a 'match' (cb)
```

Abbildung 19: Ausgabe des Egg-Skripts angewandt auf die XML-Eingabe

3.3 Optionen

Zur detaillierten Generator-Konfiguration können Optionen gesetzt werden. Es gibt *Anweisungsoptionen*, die für eine einzelne Egg-Anweisung gelten und *Skriptoptionen*, die für eine ganze Folge von Egg-Anweisungen gelten.

Syntax

Optionen müssen alle in einer Zeile stehen. Ihre Syntax lautet:

```
options := option
         | options option

option  := "[" key "=" value "]"

key     := <Optionsname>
value   := <Optionswert>
         | '"' <Optionswert> '"'
```

Abbildung 20: BNF für Egg-Optionen

Anweisungsoptionen werden nach dem Anweisungsende-Token ';' in derselben Zeile definiert. Skriptoptionen werden vor der öffnenden geschweiften Klammer '{' festgehalten.

```
.File out = "gen\\output.txt"; [append="yes"][path="absolute"]

.apply PropertyDef ( Node pProperty = </Property> ) [indent=-1] {
    int m<pProperty> = 0;
.}
```

Abbildung 21: Beispiele für die Angabe von Egg-Optionen

Optionswert

Der *Optionswert* für ein Skript oder eine Anweisung ergibt sich aus der 'nächstgelegenen' expliziten Angabe für diese Option oder ihrem Standardwert, falls keine explizite Angabe gefunden wurde. Die Suche nach der nächsten expliziten Optionsangabe erfolgt an der aktuellen Stelle selbst. Falls die aktuelle Stelle auf eine Musterdefinition verweist, wird diese befragt, ansonsten wird im statischen Skript-Kontext nach der aktuellen Stelle gesucht. Oder anders herum: die expliziten Werte aus dem Kontext der aktuellen Stelle kaskadieren quasi nach unten in die geschachtelten Musterdefinitionen.

Die Bestimmung des Optionswertes zur Option 'x' für ein Skript oder eine Anweisung 's' geht wie folgt:

1. Falls die Option 'x' in der Egg-Datei explizit für 's' gesetzt ist, dann ist der Optionswert 'value(s,x)'.
2. Falls 's' ein Verweis auf eine Musterdefinition 'p' ist und die Option 'x' explizit in 'p' gesetzt ist, dann ist der Optionswert 'value(p,x)'
3. Falls die Option 'x' in einem umschließenden Skript 'q' explizit gesetzt ist, dann ist der Optionswert 'value(q,x)'.
4. Da keine explizite Optionsangabe gefunden werden kann, wird der Standardwert für 'x' gewählt.

Egg-Optionen

Die folgende Tabelle gibt einen Überblick über alle gültigen Egg-Optionen mit ihren Werten und Einsatzkontext. In den folgenden Abschnitten werden sie näher erläutert.

Option	Werte	Standard	Kontext
fold-key	<Attribut-Name>	Name	Musterdefinition und -Anwendung
node-content	key,text, serialized	key	Musterdefinition und -Anwendung
node-var	context, iterator	context	Musterdefinition und -Anwendung
fold	name, key, path	name	Musterdefinition und -Anwendung

path	relative, absolute	relative	File-Variable
append	yes, no	no	File-Variable
keep-existing	yes, no	no	File-Variable
indent	<Integer-Zahl>	0	Musterdefinition und -Anwendung
indent-width	<Integer-Zahl>	4	Musterdefinition und -Anwendung
indent-spaces	yes, no	no	Musterdefinition und -Anwendung
inline	yes, no	no	Musterdefinition, -Anwendung, und -Skripte

Abbildung 22: Übersicht der Egg-Optionen

3.3.1 Knotenvariable Optionen

fold-key

Die *'fold-key'-Option* legt den enthält den Attributnamen fest, der als Schlüssel für ein Element fungiert. Er besagt, welches Attribut als String-Wert für ein Element verwendet werden soll (siehe auch Wert eines Knotens).

Im Beispiel in Abbildung 23 ist der Wert für den 'Column'-Knoten dann 'LEVEL'.

```
    <Column name="LEVEL"
            required="true"
            primaryKey="false"
            type="INTEGER"
    />
```

Abbildung 23: XML-Eingabe für 'fold-key' Option Beispiel

```
.apply ListColumns( Node pColumn = </Column> ) [fold-key="name"] {
      column: "<pColumn>"
.}
```

Abbildung 24: Egg-Musteranwendung für 'fold-key' Option Beispiel

node-content

Die *'node-content'-Option* hat die Werte `key`, `text` oder `serialized` und legt das Inhaltsmodell für eine Knotenvariable fest.

- `key`: Der Knotenwert ergibt sch aus dem 'fold-key' Attribut
- `text`: Der Knotenwert ist der Textinhalt des Knotens

- serialized: Der Knotenwert ist der serialisierte DOM-Baum ab dem Knoten. Leider besteht bisher noch kein Einfluss auf die XML-Formatierung des Kontenwerts.

node-var

Die *'node-var'-Option* hat die Werte context oder iterator und legt die Art der Parameterübergabe für die Knotenvariable bei einer Musteranwendung (mit 'apply') fest.

- context: Die Variable wird als einzelner Kontextknoten übergeben.
- iterator: Die Knotenvariable wird als Iterator über ihre Unterknoten angewendet. Dies ist dann sinnvoll, wenn zum Beispiel die Knotenvariable aus einer Java-Methode explizit mittels new InputNode(<parent>, <dom-node-list>) erzeugt wurde.

Bei der Anwendung einer Mustermenge (mit 'applyset') hat die 'node-var'-Option keine Bedeutung, da hierbei immer über die passenden Knoten in der Eingabe iteriert wird.

fold

Die *'fold'-Option* hat die Werte path, key oder name und legt die 'Faltungsart' fest. Die Bezeichnung ist 'historisch bedingt'. Man kann damit ebenfalls die String-Darstellung eines Knotens steuern.

Man findet nun mehrere Knoten mit einem Pfadausdruck, der String-Wert eines Kontens ist dann

- name: genau der Inhalt des 'fold-key' Attributs.
- key: wie name jedoch werden Elemente mit gleichem Namen mit einem Index-Zusatz versehen, um den Wert eindeutig zu machen
- path: Wenn man Knoten innerhalb des aktuellen Teilbaumes findet, wird aus den 'fold-key'-Attributen aller Knoten oberhalb des gefunden Knoten ein Pfadausdruck analog zu einem Pfad eines Dateisystems erzeugt.

Mit den folgenden Optionen kann man die Pfaddarstellung bei path steuern: 'fold-sep', 'index-sep1', 'index-sep2' und 'index-start'.

```
.apply NestedTables (Node pTable = </.*/table>) [fold="path"][fold-
sep="/"][index-sep1="("][index-sep2=")"][index-start="1"] {
      table-path: <pTable>.
.}
```

Abbildung 25: Beispiel-Eingabe für die 'fold' Optionen

```
      table-path: /Tabelle.
      table-path: /Tabelle/SubTabelle(1).
      table-path: /Tabelle/SubTabelle(2).
      table-path: /Tabelle/SubTabelle(3).
      table-path: /Tabelle/AndereSubTabelle.
      table-path: /Tabelle/AndereSubTabelle/Level3Tabelle.
```

Abbildung 26: Beispiel-Ausgabe für die 'fold' Optionen

Die 'fold' Option ist manchmal etwas eigenwillig zu benutzen und wird eher selten gebraucht.

3.3.2 Ausgabedatei Optionen

Die Optionen zur Steuerung des Ausgabe-Verhaltens gelten nur bei `File`-Variablen.

path

Die *'path'-Option* hat die Werte `relative` oder `append` und legt fest, ob der Dateiname relativ zum Verzeichnis der Eingabedatei oder absolut verwendet wird.

```
      .String root = "D:/src";
      .File   file = <root> + "/import.sql"; [path="absolute"]
```

Abbildung 27: Beispiel für 'path' Option

append

Die *'append'-Option* hat die Werte `yes` oder `no` und legt fest, ob die Ausgabe an eine existierende Datei angehängt werden soll, oder nicht. Dies ist zum Beispiel nützlich, um Index-Informationen aus vielen Eingabedateien zu sammeln.

keep-existing

Die *'keep-existing'-Option* hat die Werte `yes` oder `no` und legt fest, ob eine existierende Datei überschrieben oder beibehalten werden soll. Dies ist zum Beispiel nützlich, um manuell zu bearbeitende Stub-Dateien einmal initial zu erzeugen.

Beispiel

Die Verwendung der Ausgabedatei-Optionen wird im Folgenden am Beispiel einer Web-Anwendung mit verschiedenen Dialogen näher erklärt. Jeder Dialog wird in einer XML-Datei wie in Abbildung 28 spezifiziert. Daraus werden pro Dialog zwei Java-Klassen und eine Index-Datei über alle Dialoge erzeugt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dialog name="UserMgmt" path="admin">
      <fields>
            <inputfield  name="Name"    type="Text"/>
            <inputfield  name="Surname" type="Text"/>
            <inputfield  name="Street"  type="Text"/>
            <inputfield  name="ZIP"     type="ZipCode"/>
            <inputfield  name="City"    type="Text"/>
      </fields>
      <buttons>
            <button name="Reset"/>
            <button name="Update"/>
            <button name="Create" label="enter new user"/>
      </buttons>
</dialog>
```

Abbildung 28: Eingabe Datei zum FileVarDemo-Beispiel

Die eine Dialog-Klasse enthält stereotypen, generierbaren Kode, die andere enthält einen Kode-Rahmen, in dem die Dialog-spezifische Logik implementiert manuell werden muss. Letztere Klasse wird nur einmal erzeugt, falls sie nicht vorhanden ist und dann beibehalten, um die manuellen Einträge nicht zu verlieren (`keep-existing="yes"`). Die Klasse mit dem (re-)generierbaren Kode wird bei jedem Ge-

neratorlauf neu erzeugt, um Änderungen in der XML-Dialogdefinition im Java-Kode sichtbar zu machen (`keep-existing="no"`, Standardeinstellung).

Die Index-Datei dient hier zur Illustration der `append="yes"` Option. Sie wird bei jedem 'FileVarDemo'-Generatorlauf um die Liste der Datenfelder des Dialoges erweitert.

Die folgende Abbildung 29 zeigt nun den 'FileVarDemo'-Generator.

```
.generator FileVarDemo (Node pDialog = </dialog>) [fold-key="name"] {
  .boolean lWriteXRef = getArgs()[0].equals("xref");
  .String lRootDir    = getArgs()[1];
  .String lPackage = "webapp.dialog." + </path>.replace('/', '.');
  .String lDialogDir = <lRootDir> + "/webapp/dialog/" + </path>;
  .
  .if (<lWriteXRef>) {
    .apply DialogXRef() {
      .File lFile = <lRootDir> + "/webapp/field-index.txt";
                                    [append="yes"][path="absolute"]
      .apply FieldIndex ( Node pField = </fields/.> ) {
        <lPackage>.<pDialog>.<pField>
      .}
    .}
  .} else {
    .apply DialogGen() {
      .File lFile = <lDialogDir> + "/gen/" + <pDialog> + "G.java";
                                                  [path="absolute"]
      .apply DialogG(<pDialog>, <lPackage>);
    .}
    .
    .apply DialogStub() {
      .File lFile = <lDialogDir> + "/" + <pDialog> + ".java";
                             [keep-existing="yes"][path="absolute"]
      .apply Dialog(<pDialog>, <lPackage>);
    .}
  .}
  .
  .pattern DialogG ( Node pDialog, String pPackage ) {
    /*
     * Generated by 'FileVarDemo.egg' - DO NOT EDIT
     */
    package <pPackage>.gen;

    public class <pDialog>G extends Dialog {
      // <generated stuff> ...
    }
  .}
  .
  .pattern Dialog ( Node pDialog, String pPackage ) {
    /*
     * Generated by 'FileVarDemo.egg' - EDIT MANUALLY
     */
    package <pPackage>;
```

```
    import <pPackage>.gen.<pDialog>G;

    public class <pDialog> extends <pDialog>G {
      // <hand-written code geos here> ...
    }
  .}
.} // FileVarDemo
```

Abbildung 29: Beispiel für Ausgabedatei-Optionen

3.3.3 Whitespace Optionen

indent

Die *'indent'-Option* steuert den Einzug der Ausgabe. Der Optionswert ist eine Integer-Zahl mit dem Standardwert '0'. Positive Zahlen ergeben einen Tabulatorschritt nach rechts, negative einen nach links, falls dies möglich ist.

Jede Egg-Anweisung, die Musterskript enthalten kann, hat einen linken Rand, der sich aus ihrer Position in der Egg-Datei ergibt. Alle Tabulatoren in der Anweisungszeile links vom Rand des Egg-Musters werden ignoriert. Alle Tabulatoren oder Leerzeichen rechts davon werden ausgegeben.

Es folgen zwei Beispiele zur Verwendung der 'indent'-Option:

```
Zeile1
.apply AufLevel0 (...) {
      Zeile2
      .apply AufLevel1 (...) {
            Zeile3
      .}
            Zeile4
.}
Zeile5
```

Abbildung 30: Eingabe zum 'indent'-Beispiel 1

```
Zeile1
      Zeile2
            Zeile3
            Zeile4
Zeile5
```

Abbildung 31: Ausgabe zum 'indent'-Beispiel 1

```
Zeile1
.apply AufLevel0 (...) [indent=-1] {
      Zeile2
      .apply AufLevel1 (...) [indent=-1] {
            Zeile3
```

```
      .}
            Zeile4
.}
Zeile5
```

Abbildung 32: Eingabe zum 'indent'-Beispiel 2

```
Zeile1
Zeile2
Zeile3
      Zeile4
Zeile5
```

Abbildung 33: Ausgabe zum 'indent'-Beispiel 2

Für die 'generator'-, 'if'- und 'match'-Anweisungen ist der Einzug-Standardwert '-1'. Bei der 'generator'-Anweisung als äußere Klammer der Generator-Definition wird angenommen, dass die Ausgabe im Normalfall nicht eingerückt werden soll. Die 'if'- und 'match'-Anweisungen beschreiben Alternativen, die an der aktuellen Stelle in der Ausgabe erscheinen sollen. Es wird angenommen, dass in der Ausgabe keine Einrückung erscheinen soll.

Damit die Randermittlung beim Parsen der Egg-Datei richtig klappt, muss Anfang und Ende der Musterdefinition in der gleichen Spalte stehen. Dies markiert den linken Rand der Anweisung. Zusätzlich muss in den Ausgabezeilen innerhalb Musterdefinitionen muss der linke Rand beibehalten werden. Insbesondere sollten Leerzeichen und Tabulatorzeichen nicht vermischt vorkommen, da sonst die Randbestimmung durcheinander kommt.

indent-width

Die *'indent-width'-Option* legt die Tabulatorbreite in Leerzeichen fest. Der Standardwert ist '4'.

indent-spaces

Die *'indent-spaces-Option* hat die Werte `yes` oder `no` und legt die Zeichen für einen Tabulatorschritt fest. Bei `no`, dem Standardwert, wird ein Tabulator-Zeichen, bei `yes` werden 'indent-width' Leerzeichen für einen Tabulatorschritt verwendet.

inline

Die *'inline-Option* hat die Werte `yes` oder `no` und steuert, ob die Musterausgabe in einer Zeile gesetzt werden soll oder nicht. Vor und nach der letzen Ausgabezeile einer Musteranwendung ein Zeilenumbruch ausgegeben werden soll oder nicht.

Abbildung 34 zeigt die Verwendung der 'inline'-Option am Beispiel eines Generators zur Erzeugung von Java-Schnittstellen aus einer Package-Definition in XML. Dabei sollen die Parameter einer Methodensignatur in einer Zeile ausgegeben werden. Dies muss bei der Anwendung des 'Param'-Musters entsprechend beschrieben werden.

```
.generator InlineDemo ( Node pPackage = </Package> ) [fold-
key="name"] {
      .File lFile = "gen/I" + <pPackage> + ".java";
      .
      import framework.datatypes.ObjectId;
```

```
      public interface I<pPackage> {
      .apply Signature ( Node pProcedure = </Procedure> ) {

            /**
             * procedure '<pProcedure>'
             */
            public void <pProcedure>(
            .apply Param ( Node pParam = </Param> ) [inline="yes"] {
                  .String lSep  = <pParam>.
                                    getNodeIndex() == 0 ? "" : ", ";
                  .String lType = </type>.
                              equals("VARCHAR2") ? "String" : </type>;
                  <lSep><lType> <pParam>
            .} // Param
            );
      .} // Signature
      }
.} // InlineDemo
```

Abbildung 34: Beispiel zur Newline-Steuerung

3.4 Builtin-Methoden

Es gibt ein paar Utility-Methoden, die im Java-Kode des erzeugen Generators enthalten sind und in Egg-Ausdrücken oder in Anwendermethoden verwendet werden können.

Die Builtin-Methoden in der erzeugten Generator-Klasse sind in Abbildung 35 aufgeführt.

```
String[] getArgs()     // liefert die Kommandozeilen-Parameter beim Generator-Aufruf
File getInputFile()    // XML-Eingabedatei
```

Abbildung 35: Builtin-Methoden der erzeugten Generator-Klasse

Die Methoden auf Node-Variablen sind in Abbildung 36 aufgeführt. Der Zugriff auf Eingabe-Knoten erfolgt im generierten Generator über die innere `InputNode` Klasse. Sie kann Pfadausdrücke in der Eingabe 'matchen' und über die gefundenen Knoten iterierten.

```
public int getNodeCount();    // Anzahl der gefundenen Knoten
public int getNodeIndex();    // Akuteller Kontenindex innerhalb
                              // einer Musteranwendung
public Node getDomNode();     // Zugriff auf den Eingabeknoten per
                              // W3C-Node Interface
public String getContent();   // Textinhalt eines Eingabeknotens
public String toString();     // Textdarstellung eines Eingabeknotens
```

// (siehe auch 'fold-key'-Option)

Abbildung 36: Schnittstelle der `InputNode` Klasse

4 Generator-Aufruf

```
<Generator>.bat [<input-file>|@<input-list-file>]
                [-o <output-dir>]
                [arg1] ... [argN]
```

Abbildung 37: Aufruf eines Generators per Kommandozeile

5 Egg-Aufruf

```
egg.bat <egg-file> [-o <output-dir>]
```

Abbildung 38: Egg-Aufruf per Kommandozeile

6 Egg-Installation

Die folgende Tabelle 1 gibt eine Übersicht der Variablen, die in der Batch-Datei ‚egg.bat' bzw. im Shell-Skript 'egg.sh' im ‚bin' Verzeichnis gesetzt werden müssen.

Variable	Zu setzender Wert
EGG_HOME	Verzeichnis, in das der Generator-Generator entpackt wurde. Hier wird der komplette Pfadnamen mit Laufwerk eingetragen (z. B. D:\dev\egg)
JAVA_HOME	Java SDK Verzeichnis, ab Version 1.4. (z. B. D:\lib\www\java141_3)

Tabelle 1: Variablen, die zur Installation gesetzt werden müssen

egg-Dateityp

Als nächstes wird eine Verknüpfung für die Dateiendung ‚.egg' benötigt. Die Einstellung erfolgt über den NT-Explorer. Dazu das Menü ‚Ansicht/Optionen' auswählen, auf die ‚Dateitypen' Registerkarte und dann auf ‚Neuer Typ' klicken. Der erscheinende Dialog wird gemäß Abbildung 39 ausgefüllt.

Abbildung 39: Dateityp <egg> registrieren

Danach auf ‚Neu' klicken und den ‚Open' Vorgang gemäß Abbildung 40 definieren. Der Vorgang sollte ‚Open' heißen, damit die erzeugten Generatoren auch mit Doppelklick gebaut werden können.

Der Wert des <EGG_HOME> Platzhalters muss dem Eintrag aus Tabelle 1 entsprechen.

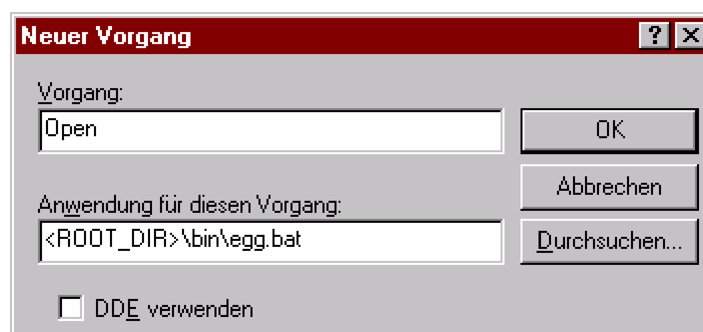

Abbildung 40: Aufruf des egg Kommandos

Nach zweimaligem ‚Ok' ist der Dateityp im Explorer registriert.

Generatoren-bau

Nun kann der ‚DialogmodellGenerator' aus dem Beispiel in Abbildung 5 erzeugt werden. Er liegt im Verzeichnis <EGG_HOME>\gen. Dazu einfach in das Verzeichnis wechseln und per Doppelklick auf die egg-Datei den Generator erzeugen. In

Abbildung 41 wird beispielhaft die Ausgabe eines erfolgreichen ‚egg'-Durchlaufs für den ‚DialogmodellGenerator' gezeigt.

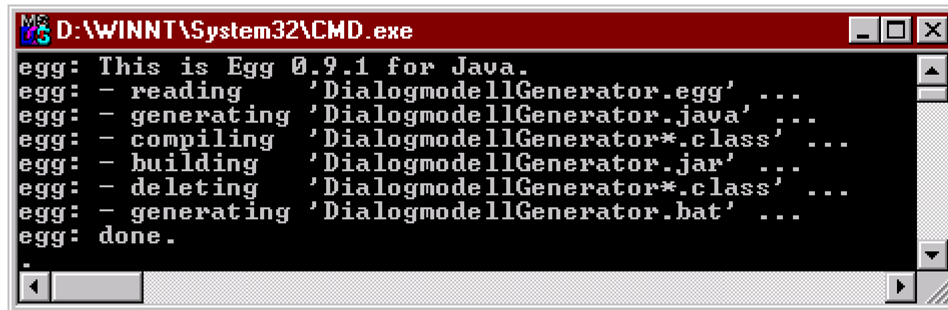

Abbildung 41: Ausgabe eines erfolgreichen egg-Laufs

Generator-Einsatz

Nachdem der DialogmodellGenerator gebaut ist kann auch die Generator-Ausgabe erzeugt werden. Dazu die ‚DialogmodellGenerator.bat' Datei mit der ‚Benutzerverwaltung.xml' Datei aufrufen. Z. B. einfach per ‚Drag And Drop' im NT-Explorer.

Es erscheint folgende Ausgabe und das Generat befindet sich schließlich in ‚BenutzerverwaltungDM.java'.

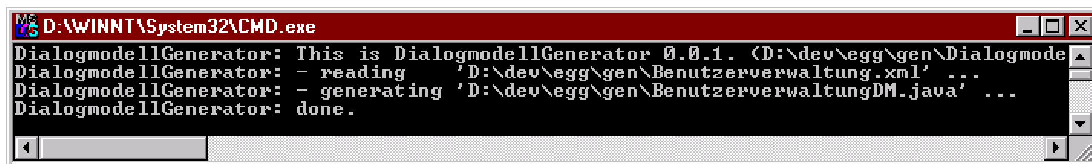

Abbildung 42: Ausgabe eines erfolgreichen Laufs des DialogmodellGenerator

Referenzen

[1] *Extensible Markup Language (XML) 1.0 (Second Edition)*;
Ort: http://www.w3.org/TR/REC-xml.

[2] *XSL Transformations (XSLT) Version 1.0*
Ort: http://www.w3.org/TR/xslt.

[3] *XML Path Language (XPath) Version 1.0*
Datei: http://www.w3.org/TR/xpath.