EGG

Ein Generator-Generator für Quellkode-Stylesheets

Alexander Knecht

26. Juli 2001

sd&m
software design & management AG
Thomas-Dehler-Straße 27
81737 München

Telefon (089) 6 38 12-0
Telefax (089) 6 38 12-150

www.sdm.de

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.
1.0	fertig	13.01.02	A. Knecht	Kosmetik.

Inhaltsverzeichnis

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 prinizipiell 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:

Warum wird ein weiters Werkzeug eingeführt und nicht XSL verwendet?

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 Pfadausfrücken 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 parameterisiert werden, um zur Generator-Definition zu gelangen.

Das Ziel ist den schon lauffähigen Prototyp-Kode dabei sowenig wie möglich zu verändern, um den Generatorbau zu beschleunigen und 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. *Muster-Definition*: Hierbei werden die A- und T-Teile in zwei Dateien ausgelagert und Redundanzen zusammengefasst. Die A-Stellen werden im T-Teil parameterisiert 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 Muster-Definitionen über Pfadausdrücke mit den gewünschten XML-Knotenmengen verknüpft und die Reihenfolge der Musteranwendung bestimmt. Desweitern 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.

Muster-Definition

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 und Abbildung 6 werden dann die Egg-Anweisungen in den parameterisierten 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 Muster-Definition 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 Muster-Definitionen 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 ensteht 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 BenuterverwaltungDM 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 ’Parameterisierte 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;
            }
      .}
```

Abbildung 5 ’Generator-Definition zum Prototypen (Teil 1)’

```
            // 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 6 ’Generator-Definition zum Prototypen (Teil 2)’

3 Generator-Definition

3.1 Zeilenmarker

Eine Generator-Definition ist zeilenbasiert. Aus den Generator-Entwicklungsprozess wird ersichtlich, dass zunächst die Generator-Ausgabe aus dem Ausgabe-Prototypen direkt übernommen wird und danach zusätzliche Zeilen eingefügt werden, um die Generator-Struktur und -Funktionalität festzulegen Die Generator-Anweisungen werden von den Ausgabe-Zeilen durch den Egg-Zeilenmarker unterschieden.

Der *Zeilenmarker* ist ein einzelnes Zeichen, mit dem alle Egg-Anweisungszeilen beginnen müssen. 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 der Punkt ‚.‘ verwendet.

3.2 Pfadausdrücke

Pfadausdrücke dienen zur Referenzierung von Knoten im XML-Eingabedokument. Ihre Syntax ist an die der regulrären 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 7 gibt die Syntax der Egg-Pfadausdrücke in der Backus-Naur-Form an:

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

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

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

Abbildung 7 '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 **Fehler! Verweisquelle konnte nicht gefunden werden.**.

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 Modifkatoren 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 garnicht.

Desweiteren 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.

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.

3.3 Anweisungen

Die die Reihenfolge der Muster-Anwendungen sowie die Transformationslogik folgt dem imperativen Programmierparadigma. Die Anweisungen in einem Muster-Skript werden nacheinander abgearbeitet. Abbildung 8 gibt die BNF für Muster-Skripte und die möglichen Anweisungen an, auf die inden 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 8 'BNF für Muster-Skripte und mögliche Anweisungen'

3.3.1 Muster-Definition

Muster-Definitionen sind direkt mit Methoden- oder Prozedur-Definitionen vergleichbar. Sie erhalten einen Namen, über den sie referenziert werden können, sowie eine Liste mit formalen Parametern, die bei der Muster-Anwendung übergeben werden und schließlich ein Skript, das die auszuführenden Anweisungen enthält.

In Abbildung 9 ist die Syntax dargestellt, die direkt von Java abgeleitet ist. Ebenso entsprechen die aufgeführten Egg-Typen den jeweilgen Java-Typen: `java.lang.String`, `java.io.File`, `org.w3c.dom.Node` und dem Basistyp `boolean`.

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

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

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

Abbildung 9 ’BNF zur Muster-Definition’

3.3.2 Muster-Anwendung

Die Muster-Anwendung ist das Gegenstück zur Muster-Definition, wie etwa der Methoden-Aufruf zur Methoden-Definition. Es gibt sie in zwei Ausprägungen, wie in Abbildung 10 ersichtlich.

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

In der zweiten Form erfolgt die Anwendung und die Definition zusammen an derselben Stelle (Inline-Definition). Eventuelle konkrete Parameter werden wie bei einer Variableninitialisierung 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" pattern-name concrete-parameters ";"

                         | "apply" pattern-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) ">"

   generator-definition := "generator" pattern-name inline-parameters "{"
                             pattern-script
                           "}"
```

Abbildung 10 'BNF zur Muster-Anwendung'

Die ‚generator-definition'-Regel aus Abbildung 10 ist der Einstiegspunkt zum Bau eines Generators. Sie muss als erste Anweisung in einer Egg-Datei stehen. Alle wieteren Anweisungen stehen dann in ihrem Muster-Skript.

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

3.3.3 Weitere Anweisungen

```
stmt-define-variable := egg-type variable-name "=" assignment-expr ";"
     assignment-expr := (<java expr code> | egg-expr)*
       variable-expr := "<" variable-name ">"

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

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

 stmt-define-version := <version string>

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

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

Abbildung 11 'BNF für Egg-Pfadausdrücke'

3.3.4 Mustermengen

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

stmt-match-patternset := "matchset" pattern-name "(" ")" "{"
                            pattern-script
                         "}"
```

Abbildung 12 'BNF für Egg-Pfadausdrücke'

4 Installation

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

Variable	Zu setzender Wert
ROOT_DIR	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 1.1. (z. B. D:\lib\www\java122_7)

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 13 ausgefüllt.

Abbildung 13 'Dateityp <egg> registrieren'

Danach auf ‚Neu' klicken und den ‚Open' Vorgang gemäß Abbildung 14 definieren. Der Vorgang sollte ‚Open' heißen, damit die erzeugten Generatoren auch mit Doppelklick gebaut werden können.

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

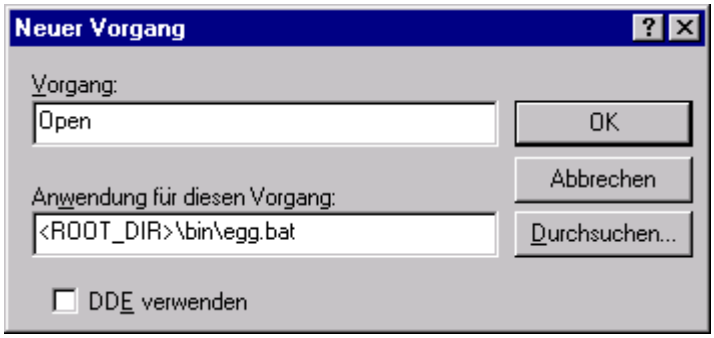

Abbildung 14 'Aufruf des egg Kommandos'

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

Generatoren-bau

Nun kann der ‚DialogmodellGenerator' aus dem Beispiel in Abbildung 5 und Abbildung 6 erzeugt werden. Er liegt im Verzeichnis <ROOT_DIR>\gen. Dazu einfach in das Verzeichnis wechseln und per Doppelklick auf die egg-Datei den Generator erzeugen. In Abbildung 15 wird beispielhaft die Ausgabe eines erfolgreichen ‚egg'-Durchlaufs für den ‚DialogmodellGenerator' gezeigt.

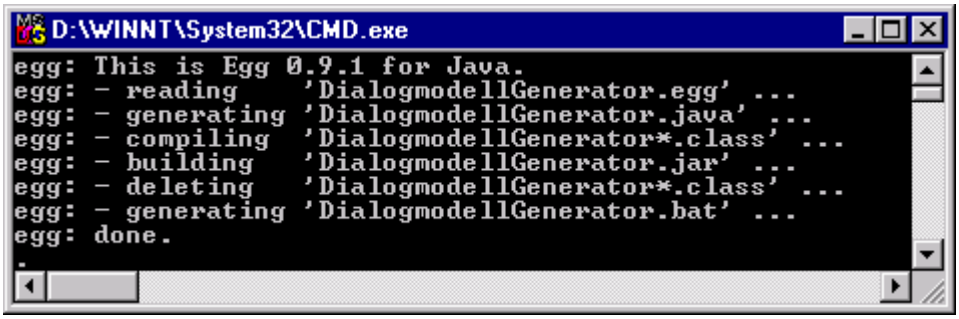

Abbildung 15 'Ausgabe eines erfolgreichen egg-Laufs'

Generator-Einsatz

Nachdem der DialogmodellGenerator gebaut ist kann auch die Generator-Ausgabe erzeugt werden. Dazu die ‚DialogmodellGenerator.bat' Datei mit der ‚Bneutzerverwalung.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'.

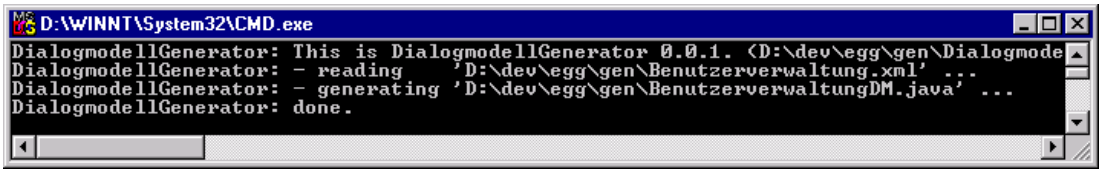

Abbildung 16 'Ausgabe eines erfolgreichen Laufs des DialogmodellGeerator'

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.