Egg

An Easy Generator-Generator for 'xml-to-text' transformations

Alexander Knecht

Translation by Matthias Nott*

June 19th, 2002

* The ideas presented in this paper have been developed by Alexander Knecht, www.generia.de. I have only translated the documentation, and I owe a lot of gratitude and respect to Alexander for putting his code in the public domain.

History

Version	Status	Date	Author(s)	Description
0.1	Working	01.06.01	A. Knecht	First Draft
0.2	Working	26.07.01	A. Knecht	Added chapter ‚Installation'
1.0	Released	13.01.02	A. Knecht	Cosmetics
1.1	Working	19.06.02	M. Nott	Translation, additions
1.2	Released	27.03.05	A. Knecht	Title adapted.

Contents

1 Motivation

This document describes the tool *Egg* for generating Generators. These generators receive an XML-file as input and write their output to text files. The transformation of the XML-contents can be freely programmed using Java. Besides Java knowledge, it is sufficient to have a fundamental knowledge about XML, and regular expressions, to create generator definitions. An understanding about the XSL idea is helpful; see e.g. [1] and [2].

The Generator definition principally corresponds to an XSL style sheet. Using path expressions, nodes in the XML source file are referenced, and their data is read. This data can be manipulated as wished, and the result can be sent to the output. This raises the following question:

Why introduce a new tool instead of just using XSL?

1. Limited programming model

   XSL has its own programming language to describe the transformations. More complex functions quickly become unreadable due to their embedding in XML tags. The standard base functions deliver only a limited set of functionality. In addition, it is not possible to define subroutines or e.g. to import library functions.

   Egg-Approach:

   The Style Sheet functionality (matching of path expressions in XML files, control of the output) is embedded in a well-known, ubiquitous programming language. Thus, all features of the programming language and –environment can be used. The learning effort is significantly reduced.

   The tool presented here uses Java as the embedding language. Other languages would likewise be possible (‘Egg for Perl’, ‘Egg for Tcl’, …).

2. No Whitespace Control

   XSL does not deliver a sufficient control over whitespaces in the output stream. Since the focus is put on the transformation of XML trees, it seems no particular emphasis was placed on the control of whitespaces in the output.

   Egg is particularly fit for building code generators. In this szenario, it is important to effectively use line breaks, indentations and tabbings where needed.

Egg-Approach:

Generally, any generated source code should be human readable and look as if it had been written by hand. This means, the optical representation of whitespaces must be maintained in the generator output, and thus it must be possible to define the whitespaces in the generator definition.

3. Generator-Output cannot be used directly as a Generator-Definition.

   XSL style sheets have to be written in XML format. This means, the generator output has to go in addition from the XSL style sheet to the XML parser of the XSL transformer. For this reason, normal text passages of the generator output must be put in CDATA sections, so that the link between generator output and representation of that output in the generator definition is no longer directly present.

   Egg-Approach:

   During the development process of (Code-) Generators, in the first place, the code that will later be generated is written by hand and tested on the target environment. This represents the prototype for the generator output, in which pattern are recognized and parametrized in order to get to the generator definition.

   The target is to alter the already working code prototype as little as possible to spead up the generator creation and to reduce errors while transforming the prototype into the generator definition.

2 Generator Development Process

The driving force for building generators is very often the wish to separate content from presentation. In the context of code generation, this means decoupling business, application specifical requirements from technical design decisions, and to allow to separately maintain both.

Methodology

To develop a generator, the following methodology has proven to be efficient. These four Stepps of a generator definition are afterwards described with an example.

1. *Prototype-Creation*: As the first Stepp, a prototype of the generator output is created and tested in the target environment.
2. *AT-Analysis*: In the output prototype, the application specific items (A) are identified and differentiated from the text areas (T) which are required for technical reasons.
3. *Pattern-Definition*: The A- (Application) and T- (Text) parts are exported to two files, and redundancies are eliminated. The A-places are parametrised in the T-part and summarised in an XML document. The parameter names from the T-part reappear as attribute names in the XML document, in order to create the reference to the A-part.

4. *Generator-Definition*: In this step, the pattern definitions are connected to the required XML node sets using path expressions, and the ordering in the sample application is defined. Moreover, the transformation logic for XML content that is eventually required for the generator output is added, and the output files are specified.

Example

In the context of a web application, each dialog is represented by a Java class which represents the bridge between the user interface and the application core. In this so-called dialog model, each dialog field has two attributes. One of these represents the dialog element on the user interface (Widget), the other saves the value which the data field contains for the application core. The dialog model class contains, in addition, a constructor which initialises the attributes, as well as getters that access the dialog fields and getters/setters for the data fields.

In the example, a generator definition is created to generate the dialog model classes.

Prototype-Creation

Figure 1 shows the prototype of the generator output for the dialog ‘Usermanagement’. Through the ‘Usermanagemenht’-dialog, the contact data of a user are managed. The dialog contains the dialog fields ‘Title’, ‘Surname’, ‚Firstname‘, ‚Street‘, ‚ZIP‘, ‚City‘, ‚Phone‘, ‚Fax‘, ‚Email‘ und ‚Organisation‘. For the creation of the dialog model prototype, only the dialog fields ‘Title’ and ‘Surname’ are implemented.

AT-Analysis

Figure 2 shows the output prototype after the AT analysis. All technical terms are marked. The T-part remains unchanged.

In figure 3, the A-terms are replaced by parameters, and redundancies created by this process are eliminated. The parameter names appear in angle brackets.

Pattern-Definition

In figure 4, the A-parts being ‚factored out’ are consolidated in an XML document. The additional dialog fields that are still missing in the prototype, are added.

Generator-Definition

Figures 5 and 6 show the formulation of the Egg-directives for the parametrized prototype from figure 3. The ‘generator’ and ‘apply’ directives define an output pattern. The syntax is similar to a Java method definition. In the parameter list of the pattern definition, the XML node is referenced using path expressions. The XML node is the context for the sample application. The pattern expressions begin with a slash ‘/’.

Inside the ‚generator’ directive, the node variable ‘Dialog’ is linked to the XML node ‘Dialog’ from figure 4. Through the ‘apply’ directives, all XML nodes that fall in the current context (‘Dialog’ node) are matched. These are e.g. the dialog fields ‘Optiongroup’, ‘Entryfield’ and ‘Selectfield’.

Inside the pattern definition, a number of variable definitions occur. On the right hand side of the assignment, the access to Egg-variables, XML-elements and -attributes are set in angle brackets. The ‚File’ variable in the second line defines the output file.

In order to now generate the dialog model generator, the generator definition is saved as ‚DialogmodelGenerator.egg’ and passed as a parameter calling the Egg tool. This tool generates a ‚DialogmodelGenerator.bat’ file which can be called with the XML file as a parameter, triggering the actual creation of the generator output which appears, in our case, in the ‘UsermanagementDM.java’ file.

```
import de.vkb.framework.datatypes.*;    // Datafield-Types
import de.vkb.framework.dialog.*;      // Dialogfield-Types

public class UsermanagementDM extends DialogModel {

    // Datafields
    private Title mTitle;
    private Text mName;

    // Dialogfields
    private Optiongroup mTitleDF;
    private Entryfield mNameDF;

    // Constructor
    public UsermanagementDM() {

        // Initialize Datafields
        setTitle(Title.EMPTY_TITLE);
        setName(Text.EMPTY_TEXT);

        // Initialize Dialogfields
        mTitleDF = new Optiongroup("Title");
        mNameDF = new Entryfield("Surname");
    }

    // Getter for Dialogfields
    public Optiongroup getTitleDF() {
        return mTitleDF;
    }
    public Entryfield getNameDF() {
        return mNameDF;
    }

    // Getter/Setter for Datafields
    public Title getTitle() {
        return mTitle;
    }
    public void setTitle(Title pTitle) {
        mTitle = pTitle;
        mTitleDF.setValue (pTitle);
    }

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

Figure 1: 'Prototype of the Generator-Output'

```
import de.vkb.framework.datatypes.*;     // Datafield-Types
import de.vkb.framework.dialog.*;       // Dialogfield-Types

public class UsermanagementDM extends DialogModel {

    // Datafields
    private Title mTitle;
    private Text mName;

    // Dialogfields
    private Optiongroup mTitleDF;
    private Entryfield mNameDF;

    // Constructor
    public UsermanagementDM() {

        // Initialize Datafields
        setTitle(Title.EMPTY_TITLE);
        setName(Text.EMPTY_TEXT);

        // Initialize Dialogfields
        mTitleDF = new Optiongroup("Title");
        mNameDF = new Entryfield("Surname");
    }

    // Getter for Dialogfields
    public Optiongroup getTitleDF() {
        return mTitleDF;
    }
    public Entryfield getNameDF() {
        return mNameDF;
    }

    // Getter/Setter for Datafields
    public Title getTitle() {
        return mTitle;
    }
    public void setTitle(Title pTitle) {
        mTitle = pTitle;
        mTitleDF.setValue (pTitle);
    }

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

Figure 2: 'Prototype of the Generator-Output following AT-Analysis'

```
import de.vkb.framework.datatypes.*;     // Datafield-Types
import de.vkb.framework.dialog.*;        // Dialogfield-Types

public class <Dialog>DM extends DialogModel {

    // Datafields
    private <Datatype> m<Fieldname>;

    // Dialogfields
    private <Fieldtype> m<Fieldname>DF;

    // Constructor
    public <Dialog>DM() {

        // Initialize Datafields
        set<Fieldname>(<Datatype>.EMPTY_<DATATYPE>);

        // Initialize Dialogfields
        m<Fieldname>DF = new <Fieldtype>("<Label>");
    }

    // Getter for Dialogfields
    public <Fieldtype> get<Fieldname>DF() {
        return m<Fieldname>DF;
    }

    // Getter/Setter for Datafields
    public <Datatype> get<Fieldname>() {
        return m<Fieldname>;
    }
    public void set<Fieldname>(<Datatype> p<Fieldname>) {
        m<Fieldname> = p<Fieldname>;
        m<Fieldname>DF.setValue (p<Fieldname>);
    }
} // end-of Dialogmodel '<Dialog>'
```

Figure 3: 'Parametrized Prototype Output'

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Dialog Name="Usermanagement">
  <!-- Dialogfields aus Prototype der Generator-Ausgabe -->
  <Optiongroup Name="Title" Datatype="Title"/>
  <Entryfield Name="Name" Datatype="Text" Label="Surname"/>

  <!-- weitere Dialogfields for die Generator-Ausgabe -->
  <Entryfield Name="Firstname" Datatype="Text"/>
  <Entryfield Name="Street" Datatype="Text" Label="Street"/>
  <Entryfield Name="ZIP" Datatype="Text"/>
  <Entryfield Name="City" Datatype="Text"/>
  <Entryfield Name="Phone" Datatype="Phone"/>
  <Entryfield Name="Fax" Datatype="Phone"/>
  <Entryfield Name="EMail" Datatype="EMail" Label="E-Mail"/>
  <Selectfield Name="Organisation" Datatype="Text"/>
</Dialog>
```

Figure 4: 'Application Specific Part as XML Document'

```
.generator DialogmodelGenerator ( Node Dialog = </Dialog> ) {
	.File outputFile = <Dialog> + "DM.java";
	import de.vkb.framework.datatypes.*;    // Datafield-Types
	import de.vkb.framework.dialog.*;       // Dialogfield-Types

	public class <Dialog>DM extends DialogModel {

		// Datafields
	.apply DatafieldDefinition ( Node Field = </.> ) {
		.String Datatype = </Datatype>;
		private <Datatype> m<Field>;
	.}

		// Dialogfields
	.apply DialogfieldDefinition ( Node Field = </.> ) {
		.String Fieldtype = <Field>.getDomNode().getNodeName();
		private <Fieldtype> m<Field>DF;
	.}

		// Constructor
		public <Dialog>DM() {

			// Initialize Datafields
		.apply DatafieldInit ( Node Field = </.> ) {
			.String Datatype = </Datatype>;
			.String DATATYPE = <Datatype>.toUpperCase();
			set<Field>(<Datatype>.EMPTY_<DATATYPE>);
		.}

			// Initialize Dialogfields
		.apply DialogfieldInit ( Node Field = </.> ) {
		 .String Fieldtype = <Field>.getDomNode().getNodeName();
		 .String Label = </Label>.equals("") ? </Name> : </Label>;
			m<Field>DF = new <Fieldtype>("<Label>");
		.}
		}

		// Getter for Dialogfields
	.apply DialogfieldGetter ( Node Field = </.> ) {
		.String Fieldtype = <Field>.getDomNode().getNodeName();
		public <Fieldtype> get<Field>DF() {
			return m<Field>DF;
		}
	.}
```

Figure 5: 'Generator-Definition for the Prototype (Part 1) '

```
            // Getter/Setter for Datafields
      .apply DatafieldGetterSetter ( Node Field = </.> ) {
            .String Datatype = </Datatype>;
            public <Datatype> get<Field>() {
                  return m<Field>;
            }
            public void set<Field>(<Datatype> p<Field>) {
                  m<Field> = p<Field>;
                  m<Field>DF.setValue (p<Field>);
            }

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

Figure 6: 'Generator-Definition for the Prototype (Part 2)'

3 Generator Definition

3.1 Line Marker

A Generator Definition is line based. From the development process of a generator one can see that in the first step, the generator output is directly copied from the output prototype. Then, additional lines are added to specify the structure and functionality of the generator. The generator directives are distinguished from output lines by the Egg line marker.

The *Line marker* is a single character with which all lines containing egg directives have to start. Whitespaces at the beginning of these lines are allowed. The ‘generator’ directive defines the line marker, i.e. the character preceding the ‚generator’ keyword is the line marker for the entire generator definition. In figure 5, the dot ‚.’ was used.

3.2 Path expressions

Path expressions serve to reference nodes in the XML input document. Their syntax follows regular expressions which are used for pattern recognition in character strings.

The Egg path expressions are kept very simple in order not to complicate unnecessarily their processing by the Egg tool. Yet, it has become apparent that more complex path expressions may well be useful in certain cases. For this reason, it makes sense to replace the Egg path expressions by XPath expressions in a later version of the tool (siehe [3]).

Syntax

The following figure 7 shows the syntax for the Egg path expressions in the Backus-Naur-Form:

```
              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 := "*"
```

Figure 7: 'BNF for Egg Path Expressions'

Semantik

A *Name of a Node* either is the name of an XML element or the name of an XML attribute. Thus, the type of the XML node is not differentiated. A *path of a node* is the path through the node tree from a context node (exclusively) down to the node itself (inclusively). For example, the node paths to the dialog fields ‘Title’ and ‘Surname’ from figure 4 relative to the root of the document tree are seen as:

- `/Dialog`
- `/Dialog/Optiongroup`
- `/Dialog/Optiongroup/Name`
- `/Dialog/Optiongroup/Datatype`
- `/Dialog/Entryfield`
- `/Dialog/Entryfield/Name`
- `/Dialog/Entryfield/Datatype`
- `/Dialog/Entryfield/Label`

The *Value of a node* is, for XML attributes, the value of the attribute. For XML elements, the node value is retrieved from an attribute that can be defined. The default attribute is ‚Name’. The default attribute and the value retrieval can be defined more in detail through a number of options.

A *path expression* now is a pattern used to describe a set of node paths. It contains single steps which can be seen as test conditions for node names. A *node path matches a path expression,* when the test conditions for all single steps are fulfilled. The *node list for a path expression* contains all nodes of which the node paths relative to a context node match the path expression.

For this pattern test, the path expression is traversed step by step, comparing each step with the node path. If the node names match, the test condition for the single step is fulfilled and the next step in the path expression can be verified.

The test for a single step can be influenced by modificators. With the NOT-operator ‘^’, the condition is met when the node names are unequal. With the ‘?’ operator, the condition becomes optional, i.e., the single step may or may not be in the node path. The ‘*’ says that the single step can appear any number of times in the node path (even not at all).

In addition, a *list of alternatives* can be given as a node name. The possibilities are encapsulated in round brackets and separated by '|'. The test condition is met, when the node name matches at least one entry of the list. Finally, the dot '.' Can be given as a *wildcard.* This matches any node name.

Examples

- `/Dialog/./Datatype`

  The node list contains all datatype attributes from figure 4.
- `/^List*/(Entryfield|Outputfield)`

  The node list contains all entry field and output field nodes in the current context of which the node path does not contain a 'List'.
- `/.*/List/.*/(Entryfield|Outputfield)`

  Matches all entry fields and output fields inside a List.

3.3 Directives

The order of the pattern application as well as the transformator logic follows the imperative programming paradigm. The directives are handled one after the other. Figure 8 shows the BNF for pattern scripts and the possible directives which are explained in the following sections.

```
    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
```

Figure 8: 'BNF for Pattern Scripts and Possible Directives'

3.3.1 Pattern-Definition

Pattern definitions can be directly compared to method- and procedure definitions. They contain a name through which they are referenced, as well as a list of formal parameters which is passed when the pattern is applied, and finally a script which contains the directives to be executed.

Figure 9 shows the syntax which has been derived directly from Java. Likewise, the shown Egg types correspond to the respective Java types: `java.lang.String`, `java.io.File`, `org.w3c.dom.Node` and the base type `boolean`.

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

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

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

Figure 9: 'BNF for the Pattern-Definition'

3.3.2 Pattern-Application

The pattern-application is the counter part to the pattern-definition, as is the method call to the method definition. There are two actual ways, as shown by figure 10.

For the first way, a preceding pattern definition is referenced using ist pattern name. Eventual parameters are passed as in a java method call.

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 Types natürlich übereinstimmen.

When passing a path expression, the parameter type can be either `String` or `Node`. For `String`-parameters, the Value of the first node in the node list is used for the path expression. For `Node`-Parameters, the pattern for each node from the node list is applied once on the path expression, passing through each node value at a time. This loop iterates only over the node variable of the pattern, i.e. fo rthe first `Node`-Parameter in der parameter list of the pattern. All further parameters are passed as normal parameters.

While executing a pattern script, there is a context node in the XML input file. All path expressions of a pattern are evaluated relative to this node. When applying a pattern, the node passed through the node variable becomes the context node of the called pattern.

For example, in figure 5 the context node in the pattern ‘DialogmodelGenerator’ is the ‘Dialog’-Parameter. In the pattern ‘DatafieldDefinition’, it is the ‘Field’-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-li
     concrete-parameter := egg-expr

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

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

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

Figure 10: 'BNF for the Pattern-Application'

The ‘generator-definition‘-rule from figure 10 is the entry point for the creation of a parameter. This directive has to be the first directive in an Egg file. All further directives then appear in their pattern script.

The inline parameter list contains only one node variable. The context node for the path expression of the `Node`-parameter is the document root of the XML entry file.

3.3.3 Further Directives

```
stmt-define-variable := egg-types 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>
                        "}"
```

Figure 11: 'BNF for Egg-Path Expressions'

3.3.4 Pattern Sets

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

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

Figure 12: 'BNF for Egg-Path Expressions'

4 Installation

Table 1 below gives an overview of the variables which have to be set in the batch file ‘egg.bat’ in the ‘bin’ directory.

Variable	Value to be set
ROOT_DIR	Directory into which the generator generator was extracted. Specify the complete path name including drive letter, e.g.

	d:\dev\egg.
JAVA_HOME	Java SDK Directory, Version 1.1 or later. (e.g. D:\lib\www\java122_7)

Table 1: 'Variables that have to be set for the Installation'

egg-File Type

The next step is to create an association for the file extension '.egg'. The setting can be done through the NT Explorer. Choose 'Tools, Folder Options, File Types' and select 'New.' The dialog box that appears is completed as shown in figure 13.

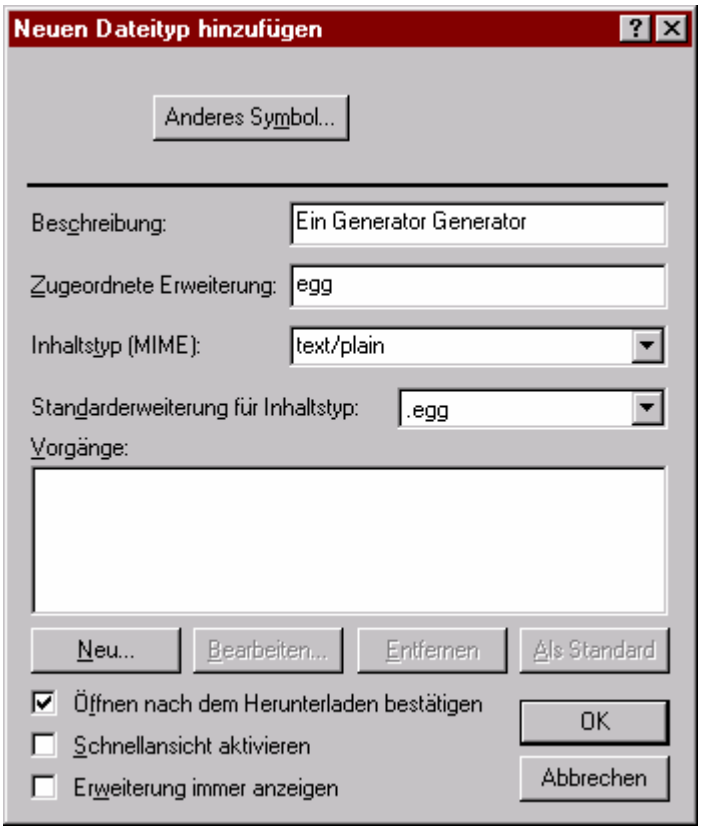

Figure 13: 'Register File Type <egg>'

Then click on 'New' and define the 'Open' procedure following figure 14. The procedure should be called 'Open' so that the created generators can be built with a double click.

The value of the <ROOT_DIR> place holder must correspond to the entry in table 1.

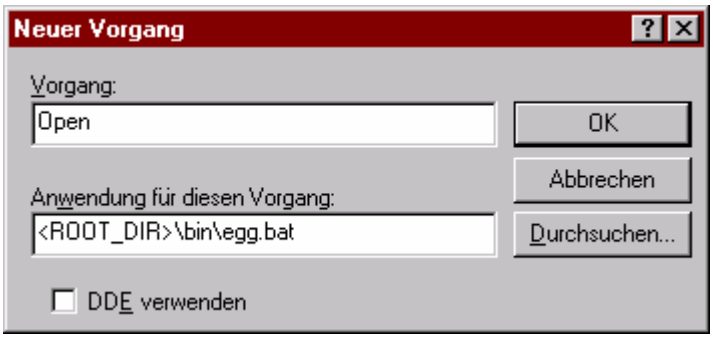

Figure 14: 'Launching the egg Command'

After pressing ‘OK’ twice, the file type is registered with the Explorer.

Generator Creation

Now, the ‘DialogmodelGenerator’ from the example in figure 5 and 6 can be generated. It can be found in the directory <ROOT_DIR>\gen. Simply change to that directory and double click on the egg file to generate the generator. In Figure 15, a sample output of a successful ‘egg’-Run for the ‘DialogmodelGenerator’ is shown.

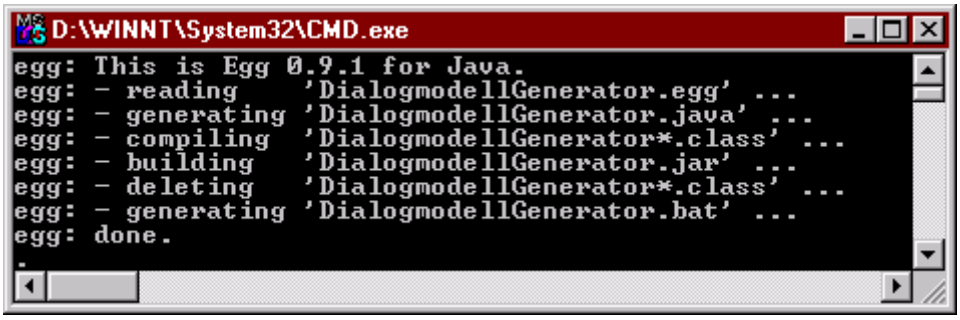

Figure 15: 'Output of a successful egg run'

Using the Generator

After the DialogmodelGenerator has been built, the generator output can be created. Just run the ‘DialogmodelGenerator.bat’ with the ‘Usermanagement.xml’ file, e.g. dragging the XML file on the batch file using the NT Explorer.

The following output will appear, and the output can be found as ‚UsermanagementDM.java‘.

Figure 16: 'Output of a successful run of the DialogmodelGenerator'

References

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

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

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

[4] *Program Generators with XML and Java*, J. Craig Cleaveland, Prentice Hall, Upper Saddle River, NJ 2001, ISBN 0-13-025878-4