

# A Proposal for a Code Generator based on XML and Code Templates <sup>1</sup>

Andreas Rausch

Institut für Informatik  
Technische Universität München  
Arcisstraße 21  
80290 München, Germany  
(gnatzm|marschal|popp|rausch|schwerin)@in.tum.de

**Abstract:** Code generation is one approach to achieve a productive industrial software production environment. In this paper we present the concepts and design of a code generator that is based on XML and code templates. The code generator is used to enable the automatic generation of Java source code describing a Componentware system from a formal specification. Using this generator it is possible to provide the core generation needs for the rapid prototyping of a Componentware system.

**Keywords:** Code Generation, XML, Componentware.

## 1 Introduction

Smaller hardware and growing systems are leading to a new degree of system complexity. For that reasons there is a need of a more industrial software production approach. Code generation is one solution to achieve this. The aim of the paper is to present the concepts and the design of an an implementation of a Generator system, able to create java source code from a formal specification. This specification consisted of an XML file representing the configuration of a Componentware system alongside a series of code template files.

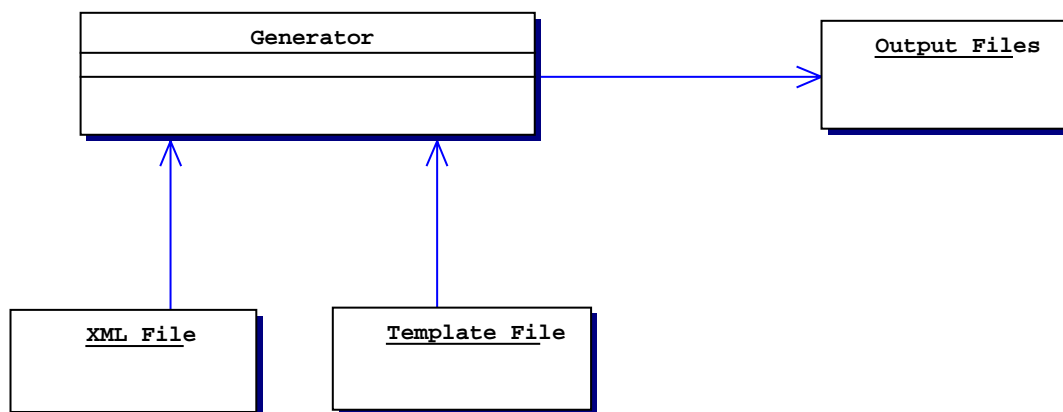


Figure 1: Overview of Generator System

---

<sup>1</sup>It originates from the research project ZEN – Center for Technology, Methodology and Management of Software & Systems Development – a part of Bayerischer Forschungsverbund Software-Engineering (FORSOFT), supported by the Bayerische Forschungsförderung.

The inputs to the system were the XML file and the Template files with the output being Java source files. An overview to the inputs and output of the generator system can be seen in Figure 8.

The approach taken in the achievement of this work package was to make all development in steps, an incremental development approach. Starting with a basic core of a file input-output system functionality was added and tested as the model was built upon. In a similar way the functional level of syntax used in the template documents was continuously increased as a means of testing the capabilities system. With every build of the system bugs and deficiencies in implementation were exposed and solved. With a full developed template developing a satisfactory out put, the work package was deemed complete.

The system achieved proved successful in the generation of java source code from the inputs given, as was the aim as stated in the package goal. The interactions within the final system are outlined in Figure 9.

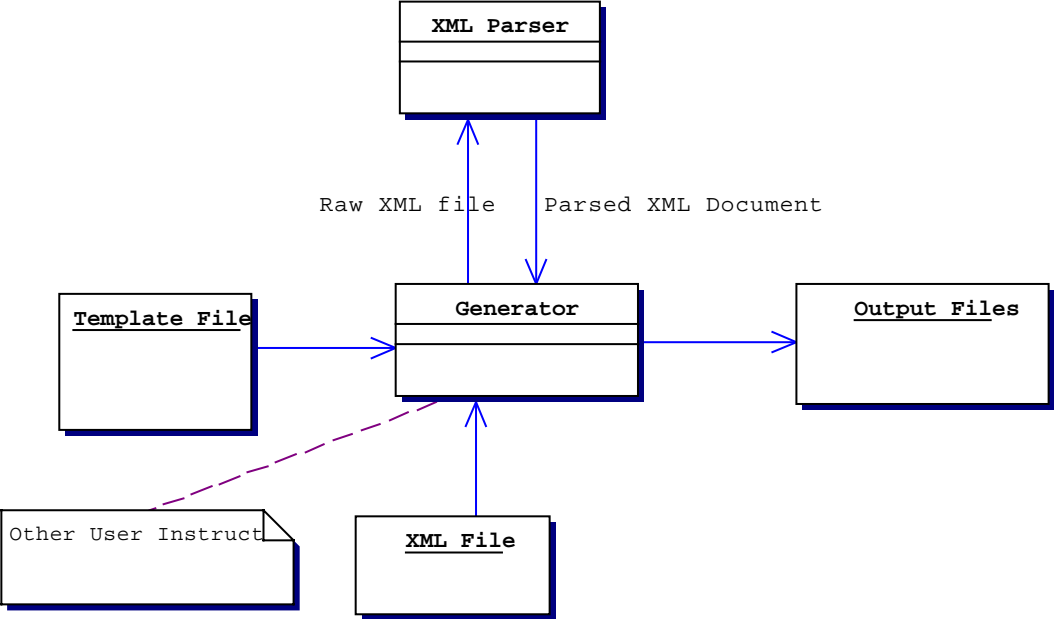


Figure 2: Interactions of Finished System

It is possible in the system to generate many files from just one input setting, with the generator spawning sub generators to cope with whatever amount, large or small, of components and interfaces required by the input documents. The complexity of the generation is shielded from the user.

A feature of the developed system was the ability to specify output paths by the user. Taking advantage of this facility it was possible to have all files created by the generator diverted to any valid file destination path. This gives the user extra control over the operation of the system.

An XML file was required for the generator to operate. This file contained all the specifics of each component in a Componentware system along with their interfaces and attributes. The processing of the file into a format navigable by the system was done externally using an XML parser. The resulting document could then be traversed like any tree like data structure.

The other main input to the system was in the form of a code template file. This included special statements contained within “#” symbols, allowing the system to take instructions during the generation process. The syntax attached to the use of these # statements allowed different actions to be performed depending on the number of #'s used. Use of the different statements enabled actions like retrieval of values from the XML document, invoking of methods or the repeated processing of an external template be performed.

Inbuilt to the developed system was a series of validity checks and error detection code used mainly in the initialisation of the system. Should a user enter incorrect instructions or invalid input files, the system would exit gracefully providing a meaningful message on the output path.

## 2 User Interface

The user interface for the system was a command line entry in the form of :

```
java designit.generator.Generator -generate <XML-file> <TPL-file> [-output_rootpath
<outputrootpath>] [-output_file_name <outputfilename>]
```

The arguments can be explained as follows:

-generate	compulsory argument used here for completeness. In the future different commands may be possible in its place.
<XML-file>	compulsory argument being a string representation of a path to an xml file
<TPL-file>	compulsory being a string representation of a path to a template file
[-output_rootpath <outputrootpath>]	optional argument being a path to a folder where is is wished that all the output from the generator will be dumped. This folder is to be created if not already on existence
[-output_file_name <outputfilename>]	optional argument allowing a standard general output file to be named. This file will be saved at the default root path location if it has been set

If an output file name location was set in a template file it was this location that was used, overriding the command line default file name.

By not specifying an output file name location in either the command line or the template, a runtime error would occur and the generate would exit.

Related to the user interface were the files supplied by the user, in the form of an XML file and a template file. To explain these files and their format, it would be useful to present a small sample of input files and generated output. This name given to this small sample was the intro sample.

Code was generated by using the combination of the standard template files, which described the code of interfaces, components and attributes, alongside an XML file describing the specifics of the system to be produced. Firstly an example of an XML file is shown below in Text Extract 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v3.0 NT (http://www.xmlspy.com) by Andreas Rausch
      (4Soft GmbH) -->
<!-- edited by Andreas Rausch -->
<!DOCTYPE COMPONENT_SPECIFICATION_DOCUMENT SYSTEM
      ".\..\..\generator\DesignItSpecification.dtd">
<COMPONENT_SPECIFICATION_DOCUMENT>
  <COMPONENT_SPECIFICATION NAME="designit.sample.intro.CB"
    MIN_CARDINALITY="1" MAX_CARDINALITY="1">
    <INTERFACE_SPECIFICATION NAME="IB" MIN_CARDINALITY="1"
      MAX_CARDINALITY="1">
      <CONNECTION_SPECIFICATION NAME="IA_2_IB">
        <CONNECTION_END NAME="FROM_IA_2_IB" TYPE="Set(IB)"
          MIN_CARDINALITY="1" MAX_CARDINALITY="*" />
      </CONNECTION_SPECIFICATION>
      <MESSAGE_SPECIFICATION NAME="bar">
        <BEHAVIOR_SPECIFICATION
          BEHAVIOR_SPECIFICATION_LINE="System.out.println(Thread.curren
            tThread().getName()+ &quot;; &quot; + &quot;method bar called
              on interface IB:&quot; + getContext().getName());"/>
        </MESSAGE_SPECIFICATION>
      </INTERFACE_SPECIFICATION>
    </COMPONENT_SPECIFICATION>
  </COMPONENT_SPECIFICATION_DOCUMENT>
```

```
</COMPONENT_SPECIFICATION>
</COMPONENT_SPECIFICATION_DOCUMENT>
```

*Text Extract 1 Example of XML File-Used In Intro Sample*

The XML file represented perfectly the hierarchical structure of the system to be generated, with components on top, components having child interfaces and then interfaces having child attributes and connections. The specifics of the code relate to the template file, an example of which is given in Text Extract 2. These specifics are beyond the scope of this report.

```
##FileGenerator.setOutputFile(
    ##JavaGeneratorHelper.getFileOutputPathFromComponentName(
        #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
        ttributeNAME#)##)##
/*****
* this file is generated by the DesignIt code generator          *
* do not edit it!                                              *
* in case of re-generation changes may be lost!                *
*****/
package ##JavaGeneratorHelper.getPackageNameFromFullName (
    #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
    ttributeNAME#)##;

import designit.engine.*;

public class ##JavaGeneratorHelper.getClassNameFromFullName(
    #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
    ttributeNAME#)## extends Implementation
{
    public ##JavaGeneratorHelper.getClassNameFromFullName(
        #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
        ttributeNAME#)##(Wrapper myWrapper)
    {
        super(myWrapper);
    }

    /* method setValue should never be called on a component. */
    public void setValue(Object value)
    {
        throw new java.lang.RuntimeException("method setValue should never be
            called on a component!");
    }

    /* method getCopyOfValue should never be called on a component. */
    public Object getCopyOfValue()
    {
        throw new java.lang.RuntimeException("Method clone should never be
            called on a Component!");
    }
}
```

*Text Extract 2 Example of Template File-Used In Intro Sample*

It can be seen that the template file closely resembled an example of a normal Java source file. The template file can be conceptualised as the “frame” onto which the substance contained in the XML file was put. It was a mixture of regular straight Java and special statements.

During the generation process the template file was read through, line by line, by the generator. For normal text the generator dumped the line straight into an output file. In the case of special statements being read the generator must evaluate the output itself. These special statements, contained between ### markers, allowed the generator to call methods or make references to the XML document and

then use the resulting information to determine what should be written to the output file. The resulting generated component CB can be seen below in Text Extract 3.

```
/*
*****
* this file is generated by the DesignIt code generator
*
* do not edit it!
*
* in case of re-generation changes may be lost!
*
*****
*****/
package designit.sample.intro;

import designit.engine.*;

public class CB extends Implementation
{
    public CB(Wrapper myWrapper)
    {
        super(myWrapper);
    }

    /* method setValue should never be called on a component. */
    public void setValue(Object value)
    {
        throw new java.lang.RuntimeException("method setValue should never be
called on a component!");
    }

    /* method getCopyOfValue should never be called on a component. */
    public Object getCopyOfValue()
    {
        throw new java.lang.RuntimeException("Method clone should never be
called on a Component!");
    }
}

```

*Text Extract 3 Example of Generated Code-CA.java from Intro Sample*

The generated code is fully compileable and fully functional. It can be seen that the generator has removed all the special statements and replaced them with the correct text for this particular component, whilst ignoring the regular text. Under normal circumstances this is last interaction that the user will have with the generation system.

### **3 Generator Package in Detail**

These three classes comprised the generator package. The overall entry by the user was made through the Generator class itself. This was the class that provided all the initialisation checks and operations. With all being present and correct the generator would allow the actual detailed generation process to begin. An instance of FileGenerator was created and a method to begin processing the first template called. The FileGenerator could then take control, spawning new FileGenerators as required to divide up its workload of generation tasks. The JavaGeneratorHelper class contained a number of helper methods which were accessed by templates through the FileGenerator. These methods performed operations only relevant to templates being used to generate code for Java Systems.

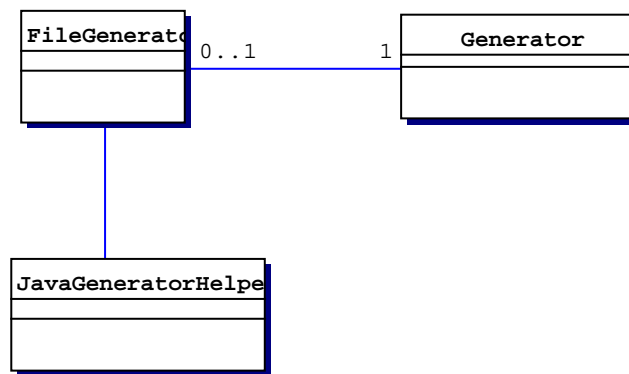


Figure 3: Classes in the generator package.

### 3.1 Generator

In the Generator class we placed a main method, the only occurrence of a main method in the entire generator package. The initiation of the generation process was triggered from here, before which a series of file validity checks etc. were performed.

The user inputs to the system were made through this class, being provided as a series of parameters as specified in section 2. This was the only interface with which the user could interact with the system.

All methods in this class were static. This choice was a deliberate one, allowing the methods of this class to be accessed without either having a new instance of the object or having a back pointer to an already created object. Finally Generator was also the storage place for any generation wide variables and data structures.

### 3.2 File Generator

File Generator was the central processing area of the entire system. It was here that all input streams were processed and output streams written to. In the FileGenerator an input was read (initially direct from a template) line by line. This line of text can contain the following 4 types of information.

- a statement enclosed by a # statement
- a statement enclosed by a ## statement
- a statement enclosed by a ### statement
- either plain text or a combination plain text and one of the statements above.

Using the information contained in the special statements the FileGenerator decides on an action to be performed, for the # a method call is made, for the ## a lookup is made to the XML file and for ### a high level execution decision is made. A full description of the statements follows in the next sections.

The hierarchy for hash recognition in the File Generator input reader was Triple hash then Double hash then Single hash. In a line containing multiple instances of special statements, the resulting text or operation is determined in the above order.

#### 3.2.1 One Hash Statement

A # statement indicated a reference is being made to the underlying XML document.

There can be more than one # statement in a line, each statement being evaluated one after the next.

A # statement was the last statement in a line to be processed, the other statements having precedence

over it.

Shortcuts could be created which would allow certain text in a # statement be replaced by a corresponding value if it existed in the Generator hashTable.

The syntax for a single hash statement was as follows:

*single hash statement* = *path\_to\_base\_element* + "." + */"getAttribute"//"getElements"/* + *key\_to\_sub\_search*

path_to_base_element	Names of a series of nodes (dot separated) which defined a path to an element. This path started at the top of a parsed XML document and ended at the desired element.
GetElements	statement which lead the evaluator to return a series of cloned documents. Each cloned document was a clone of the chief xml document, modified to only have one instance of a child of the base element of name contained in key_to_sub_search.
GetAttribute	statement which will lead the evaluator to return a value contained by an attribute. This attribute will be an attribute of the base element of the name contained in key_to_sub_search.
key_to_sub_search	A string which will be used by getElements/getAttribute as an evaluation key.

### 3.2.2 Two Hash Statement

A ## statement indicates that a method call is to be made through the Java Runtime Environment. The method call between the hashes is made in the normal way as per normal in Java.

For operational reasons all methods must rest in the designit.generator package.

The system at the moment is configured to recognise nested ## and will not recognise consecutive ## statements.

The rules for a double hash statement are as follows:

It is assumed that there is only one double hash statement per line with no other single or triple hash statements following it.

The parameters for the method can consist of

- -any number of parameters
- -any parameter in the form of a # statement.
- -any parameter in the form of a ## statement.
- -any parameter in the form of normal text / normal text + # statement / ## statement.

### 3.2.3 Three Hash Statement

The purpose of the triple hash statement in this system is to allow the systematic processing of a number of similar elements.

By using the "EXECUTE\_WITH" keyword another template will be processed in association with an array of modified documents evaluated through the getElements.

This template is specified by the text immediately preceding the statement.

Effectively for one use of a triple hash statement you can generate any number of components on the same level of a document.

A feature of the triple hash statement is the ability to create navigation shortcuts. The shortcut is

represented by a key and value in the added to the system wide hashtable stored in the Generator class.

The key to be added is the string contained between the first single hashes in the opening for statement.

The value added is the string contained in the second single hash statement, minus the "getElements" command statement.

Wherever the key is encountered in the execution of a generation it is replaced by its accompanying value.

At the closing triple hash the key / value are removed from the hashtable and are not available for future use as a shortcut.

It is also allowed that within a definition of a key or value that an already existing shortcut can be used. The syntax for a triple hash statement is as follows:

*triple hash statement = opening\_triple\_hash + |execution\_statement| + closing\_triple\_hash*

opening_triple_hash	"###FOR #" + shortcut_key + "# IN #" + shortcut_value + "###"
shortcut_key	string to be added to hashtable as the key.
shortcut_value	Dual purpose statement representing text to be added as value to hash table and also defining a single hash statement to be evaluated for an array of modified documents.
execution_statement	"###EXECUTE_GENERATION_WITH"+template_for_generation+"###" .
template_for_generation	String representing the file location of a template file to be used in the creation of source code - one file being created for each modified document returned by opening_triple_hash.
closing_triple_hash	"###ENDFOR #" + shortcut_key + "#".

### 3.3 JavaGeneratorHelper

The JavaGeneratorHelper class existed to provide methods for use in the generation of Java Source Code. These methods were accessed by text contained in a template, and called through the template processing system in the FileGenerator. Only templates used in the generation of Java code would reference any method in this class.

## 4 Conclusion and Further Work

The presented concepts and the design of the code generator based on XML and code templates have been successfully implemented and used in a concrete project. However, some limits and drawbacks appeared, that may be fixed in future.

Obviously, the syntax in the templates causes some problems, as we are not able to distinguish between opening and closing hashes. For that reasons we will have to extend the syntax and the generator. Moreover, there is an additionally need for a condition statement within the templates, loops are not enough. Hence, we again have to extend the syntax of the templates. Finally, the generator has to be used in more projects, the documentation and the source code has to be provided as open source.