CODE GENERATION USING XML BASED DOCUMENT TRANSFORMATION

Soumen Sarkar
sarkar_soumen@yahoo.com

Craig Cleaveland
craig@craigc.com

Published on

TheServerSide
YOUR J2EE COMMUNITY

Abstract

Software projects can greatly benefit from custom document generators. The term 'document' in the context of a software project includes various software artifacts. A software project consists of various activities which lead to the production of documents in many forms. An example of such a document is a programming language file that is used to implement computational specifications. The document transformation approach, when applied to the particular area of code generation, incorporates a higher degree of automation into the software development life cycle. The benefits of this approach are model driven programming, automatic update propagation, and a higher degree of consistency enforced by a generative approach.

With recent interest in XML standards and availability of tools supporting these standards, it has become possible to generate multiple types of documents by applying XML document transformation technology. Using XML document transformation technology, it has become easier to develop custom code/document generators in application development projects.

An object based server side infrastructure was used in the 'sample' project referred to in this whitepaper. Object and relational model code generation coupled with object services provided by the EJB framework created a very powerful paradigm of server side infrastructure development. The project had a tremendous lead by being able to build further on this sophisticated server side infrastructure rather than spending time on building the infrastructure itself. The project was totally focused on building application logic and delivering functionality. Out of approximately 2300 java files in the project, 1900 files were generated using XML document transformation technology.

Introduction

This paper is mostly about how code generation aids speedier application software development. However, this paper will also highlight the fact that source code generation processing is a particular application of the broad technology of XML based document transformation.

Document generation is an old concept in software engineering. In our day-to-day activities in software development, we use document generators in the form of programming language compilers, program generators, and html document generators to name a few. However, software projects also rely on the availability of professionally developed tools to achieve generation in some fixed areas. Professionally developed tools may not be free and may not be suited to the custom requirements of a software project. On the other hand, existing approaches to generation from custom language specifications are not easy to implement.

In the opinion of the authors, complexities in the current approach to code/document generation have impeded widespread acceptability of the generative approach in application software development projects. This article will show how current XML based standards and freely available tools supporting these standards provide an easier way to develop custom code/document generators.

A real world software project has greatly benefited by following the generative approach in application software development. The purpose of the project was to develop a network management system for a large network. The project used generation extensively in the areas of Enterprise Java Bean code, SNMP based network management code, Java helper utilities, and the SQL schema for relational databases. Though the project did not undertake, it was stillpossible to generate other kinds of documents such as SVG pictures or PDF documents out of XML data the project had.

This paper is organized into four sections, namely,

- Principles for document/code generation
- An outline of the XML approach
- Details of the XML approach
- The generative approach in application software development projects

While this paper will demonstrate that an XML based generation approach is easy enough for it to be considered in many projects, it is not a general approach. There are some limitations. The conclusion addresses this and puts the applicability of the current XML transformation approach into proper perspective.

Principles for document/code generation A code generator generally consists of a number of distinct processing phases as shown in the following figure.

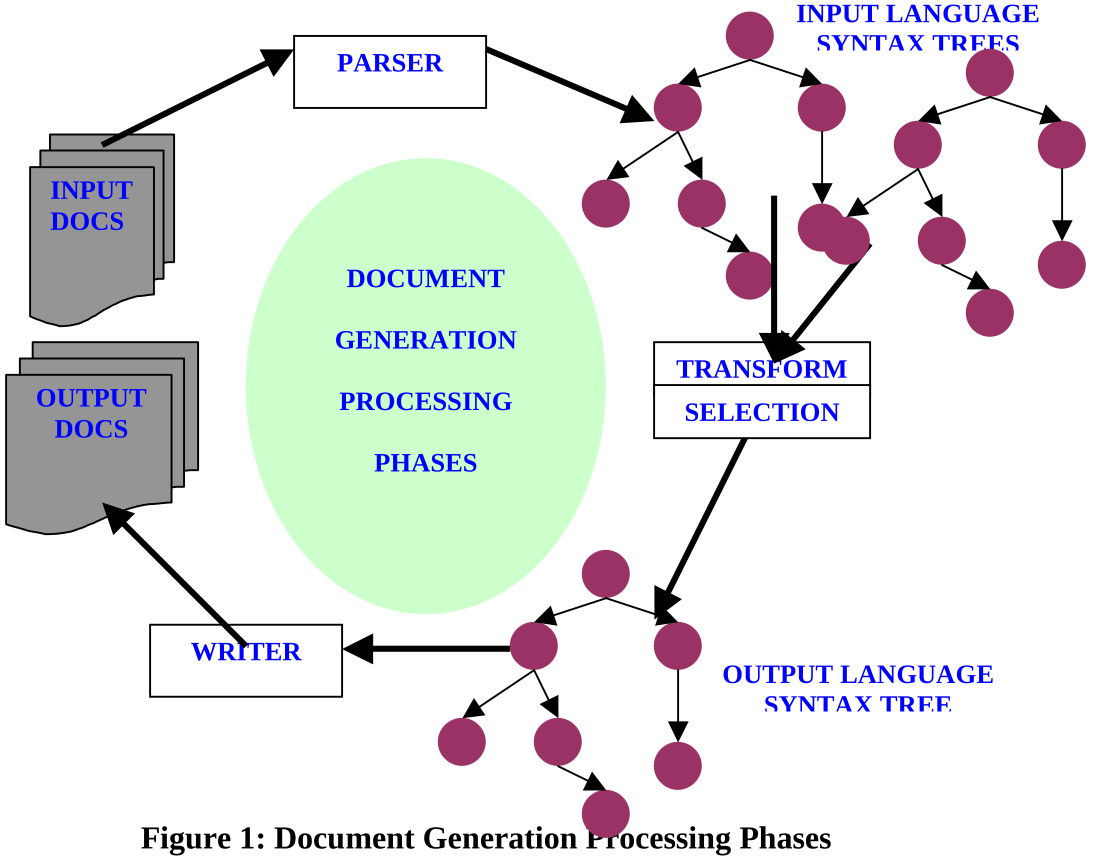

Figure 1: Document Generation Processing Phases

Input files contain specifications of some sort conforming to the syntax of the input language. Input files may be related to each other, for example, the **#include** directive in C or the **import** directive in the Java.

The parser, which could be hand written or generated by some tool, parses the input language documents token by token and constructs the input language syntax tree if the input language conforms to all syntax rules as specified by the grammar. In case of an error, it is the responsibility of the parser to indicate it at the precise location in the document. Two kinds of parsers exist, namely, the **tree constructing parser** and **the event driven parser.** Tree constructing parsers parse the input document in order to construct the tree, and hand it over for further processing. Event driven parsers notify the user as soon as desired grammatical constructs are recognized. In event driven parsing, the user needs to determine which parse events to listen to and which ones to ignore. The user constructs the data structures as parse events are being received. .

The parse tree holds in-memory representation of the input language documents. Construction of the parse tree is absolutely essential since it captures the document in a data structure on which all further processing algorithms operate. Note that there can be more than one tree representing documents conforming to different languages. For example, Java byte code could be generated from multiple languages including Java, and there could theoretically be a processor accepting all these languages. This paper shows an example where application of multiple languages (in a very loose sense) is necessary.

Once the parse tree has been formed, two kinds of processing may occur, namely, **transformation** and **selection.** The goal of the transformation phase is to convert a given tree into another tree based on some goals. For example, optimization processing might transform the parse tree into a simpler tree by eliminating redundant information. The selection phase is more important in the sense that it drives the code generation based on the information found in the tree. For example, we evaluate an expression like (a+b) by selecting the + operator and then browsing the tree rooted at the + operator to understand that variable a and b should be added. In effect, selection means browsing through the tree and driving the code generation.

The **writer** operates on the output language syntax tree to produce the output language document and write it to the file system. The writer is the opposite of the parser in the sense that it produces the language document from the tree as per the output language grammar. Bear in mind that the output language document may be of the same or of a different language.

An outline of the XML approach

In the XML approach, the input language is always XML; however, there is no restriction on what the output can be. XML has a fixed grammar, which has enabled the development of excellent parsers.. XML, with its element, attribute, and namespace capability can cover the specification requirements of many domain languages. The best testament to this fact is the availability of so many domain specific languages that are based on XML. From this standpoint it could be argued that XML is a very good base language to define domain specific languages. For example, consider that Java does not have an enumeration type. A project using Java may use the following XML specification to generate Java utility classes having enumeration functionality:

```
<EnumDef name = ‘DayOfWeek’>
   <choices>
      <choice name = 'SUNDAY' value = '0' />
      <choice name = 'MONDAY' value = '1' >
      ……
      <choice name = 'SATURDAY' value = '6' />
   </choices>
</EnumDef>
```

The above specification is intuitive enough to convey the fact that `DayOfWeek` is an enumerated type with seven distinct values. This also shows that the process of defining a domain specific language in XML consists of defining the markup elements and element attributes. In the case where domain language elements clash with preexisting XML elements, the XML namespace facility should be used to distinguish domain language elements. The set of XML namespace, elements and tag attributes define the XML based domain language.

Once the input language document is defined in XML, XSLT scripts can be written to process documents conforming to the input language and generate output documents in various forms. As previously explained, a document could be a source code file or it could be an electronic document in various forms. XSLT is an XML based language standardized by W3C. Processors supporting the XSLT language standard are used to translate XML documents into other XML or text documents. There are a number of free XSLT processors available; this study used the SAXON XSLT processor developed by Michael Kay.
XSLT is used for document transformation; for example, it can be used in a multiple line publishing scenario including HTML for web clients, Wireless Markup Language (WML) for wireless clients, and Portable Document Format (PDF) for print documentation. The concern of one document content yet multiple presentation formats is neatly addressed by using XSLT transformation in the web based information architecture. However, in our opinion, the preceding use of XSLT has received the most attention in web document publishing scenarios and not for its utility of custom code/document generation in application software development projects. This paper emphasizes the fact that XSLT offers an easy approach to code generation.

Please note that this strategy of code generation is perfectly in accordance with the processing model in figure 1. Compared to the traditional approach of code generation using **lex** and **yacc**, this approach has many features aiding the faster development of code generators as shown below:

Parser: XML document parsing is implemented in the XSLT processor, whereas with a custom parser, one needs to implement the parser or understand how to use a program generator such as lex/yacc to generate the parsing framework.

Tree: The XSLT processor constructs the tree and provides access to the tree as per the XPATH specification. The user does not have to bother constructing the tree at all. With the custom parser, the user needs to populate the tree as the parsing progresses.

Tree processing: The XSLT processor provides access to the tree through the XPATH expression and provides many programmatic constructs and functions to perform tree processing. In other words, XSLT users write XSLT scripts to perform operations on the tree. On the other hand, this part needs to be programmatically implemented by code generator writers following other approaches. XSLT programming for code generation programming is at a high level, namely, at the level of tree abstraction. Note that with XSLT, eventhough the code generation programming is at the tree level abstraction, the programmer never needs to worry about tree data structure implementation details.

Writer: The XSLT processor implements this part too.

To write a code generator, the software developer only writes XSLT scripts using XSLT and XPATH facilities. Another important benefit of XSLT based code generation is the amount of flexibility that is allowed in the change of input document grammar.More specifically, additional elements and attributes can be introduced in the input language specification without affecting code generation scripts. Contrast this to a custom parser driven approach, where there will be major change propagation throughout the code generation system. These are complexities in the application software development of custom code generation not present in the use of XSLT.

Figure 2 shows the process of document generation using XSLT. Typically, XSLT processors produce one output file. An external utility could break the single output file into multiple files. For example, in the case of source code, the output file can delimit the beginning and the end of each file with markers not likely to occur as part of the source code. Furthermore, the location of each file could also be indicated as part of the generation. The external utility then processes the single output file to produce multiple files in multiple locations. Similarly, input to the XSLT processor should preferably be one XML file including other XML files. There are techniques to achieve **XML include** and W3C is working on a standardized **XML include** mechanism.

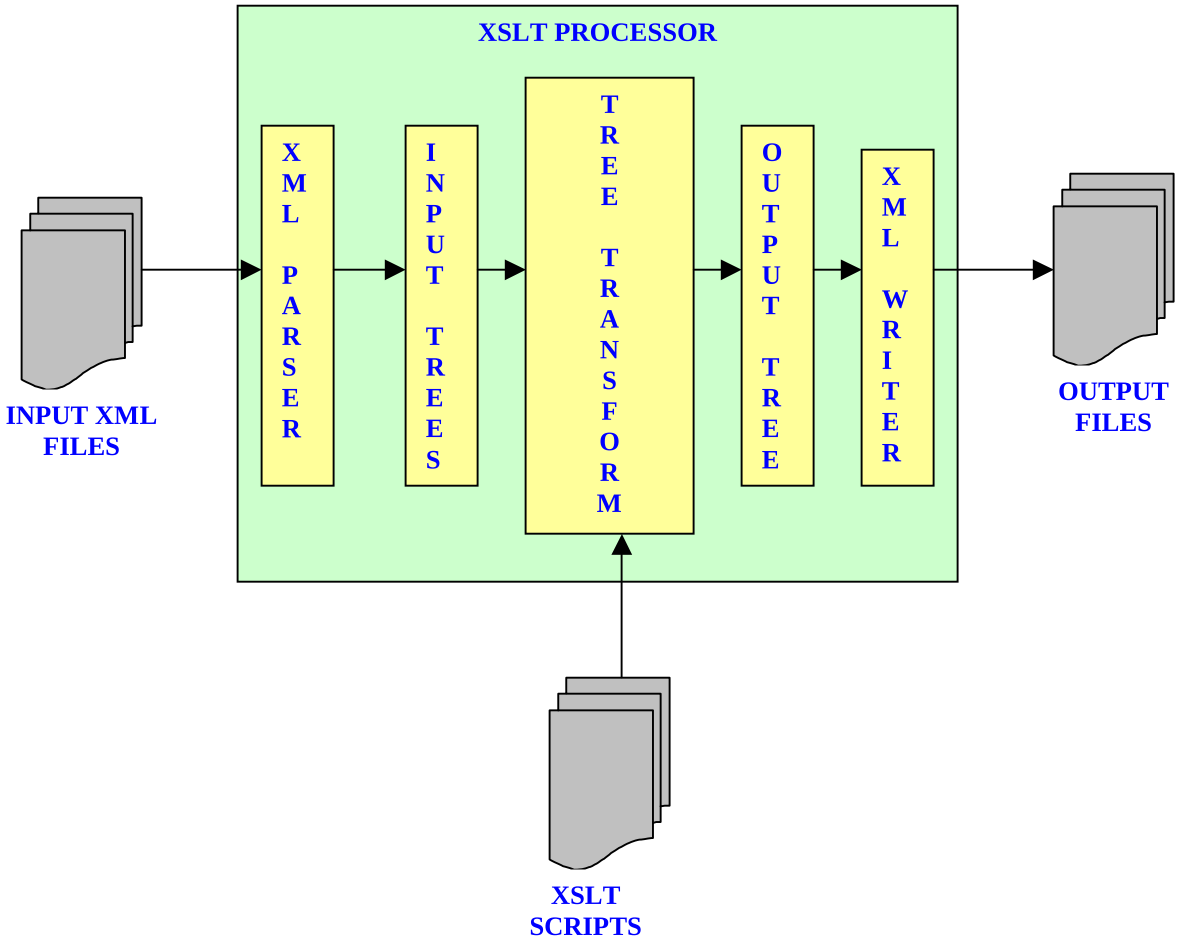

Figure 2: XSLT based document generation process

Details of the XML approach

The detail lies in the use of XSLT and XPATH to implement the desired transformation from the XML source document to the destination document. XPATH is a separate W3C standard that is intimately related to XSLT. XSLT defines a transformation language that operates on the tree representation of XML documents. XPATH defines the syntax of a language that is used within XSLT scripts to address parts of the XML document tree being transformed. XPATH is the ‘tree addressing’ language. An XSLT processor implements XSLT and XPATH specifications and, optionally, some extension functions to make the life of an XSLT script-writer easier. Accordingly, code or document generation using XML/XSLT requires the following abilities:

- **Ability 1**: The ability to define the application’s specifications in XML. This involves the determination of XML elements, XML element attribute names, values, and the nesting structure.
- **Ability 2**: The ability to visualize the tree that will result from the input XML documents containing code generation specifications.
- **Ability 3**: The ability to use XSLT and XPATH facilities to browse through the XML tree and generate output based on the tree content.

It is beyond the scope of this paper to explain XSLT and XPATH. There are a number of well-written books and internet resources available on these subjects. Installation and usage details of the XSLT processor are very easy. This section demonstrates the concepts covered so far in two ways, namely:

- Through a very simple code generation example. The example will illustrate the need for the three required abilities as defined above.
- By Describing the various areas of code generation in a real world software project.

Example

Let us demonstrate the Java enumeration utility class code generation. We first need to decide how we would like to specify the code generation (ability 1). We decided that a particular enumeration specification will look like the following:

```
<EnumDef name = ‘name of the enum’>
<choices>
<choice name = ‘name of the choice’ value = ‘integer value’/>
</choices>
</EnumDef>
```

The specification has the following statements, not expressible formally with XML syntax:

1. `<EnumDef>` element attribute `name`, `<choice>` element attribute `name` can only have values as per Java language specifications for variable names. This is due to the fact that the target language is Java.
2. `<choice>` element attribute `value` can only have distinct integral values. There has to be at least one `<choice>` element nested within the `<choices>` element.

With this specification in mind, we start defining XML documents containing Java enumeration code generation specifications. We have two documents shown below. As the names hint, `AuthoredEnum.xml` was authored by project software developers, whereas `GeneratedEnums.xml` was generated by a utility program from some data source.

AuthoredEnums.xml	GeneratedEnums.xml
`<Enumerations>`	`<Enumerations>`
`<EnumDef name = ‘AccessEnum’>`	`<EnumDef name = “ActionEnum”>`
`<choices>`	`<choices>`
`<choice name = ‘read_only’ value =`	`<choice name = “start” value =`
`‘1’/>`	`“1”/>`
`<choice name = ‘read_write’ value =`	`<choice name = “stop” value = “2”/>`
`‘2’/>`	`<choice name = “test” value = “3”/>`
`</choices>`	`</choices>`
`</EnumDef>`	`</EnumDef>`
`<EnumDef name = ‘SeverityEnum’>`	`<EnumDef name =`
`<choices>`	`“AccessControlEnum”>`
`<choice name = ‘critical’ value =`	`<choices>`
`‘1’/>`	`<choice name = “permit” value =`
`<choice name = ‘major’ value =`	`“1”/>`
`‘2’/>`	`<choice name = “deny” value = “2”/>`
`<choice name = ‘minor’ value =`	`</choices>`
`‘3’/>`	`</EnumDef>`
`<choice name = ‘warning’ value =`	`</Enumerations>`
`‘4’/>`	
`</choices>`	
`</EnumDef>`	
`</Enumerations>`	

Figure 3:

One technique of XML inclusion (using an external parseable entity reference) is used to present only one XML input file at XSLT processing time. The XML file in effect has all the enumeration definitions. The XML file is shown below:

AllEnums.xml

```
<?xml version = "1.0"?>
<!DOCTYPE AllEnums[
<!ENTITY include_authored_enums SYSTEM "AuthoredEnums.xml">
<!ENTITY include_generated_enums SYSTEM "GeneratedEnums.xml">
]>
<AllEnums>
&include_authored_enums;
&include_generated_enums;
</AllEnums>
```

Figure 4: Technique for XML include

The second step is to visualize the internal tree structure that will be created inside the XSLT processor. The tree structure is depicted in two parts. The first part shows the overall tree structure and the second part shows the tree structure rooted at the first `<EnumDef>` element.

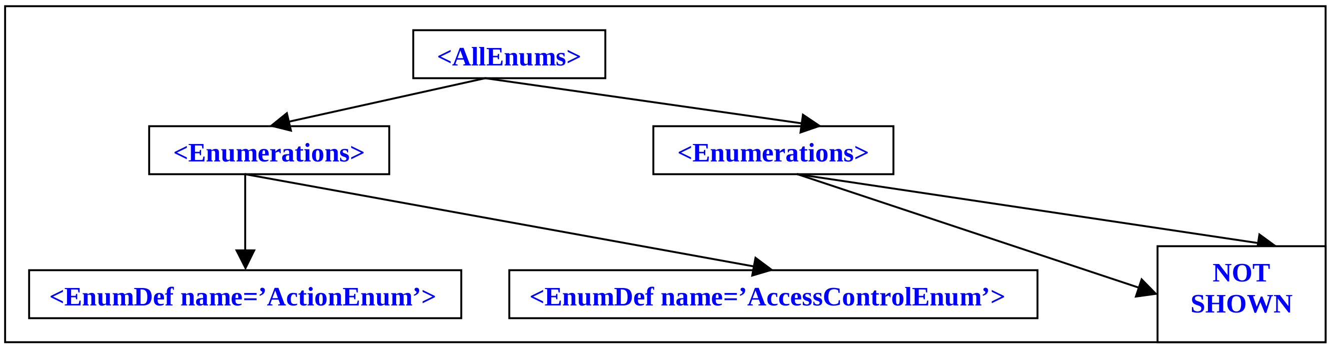

Figure 5: Part 1 of the tree structure

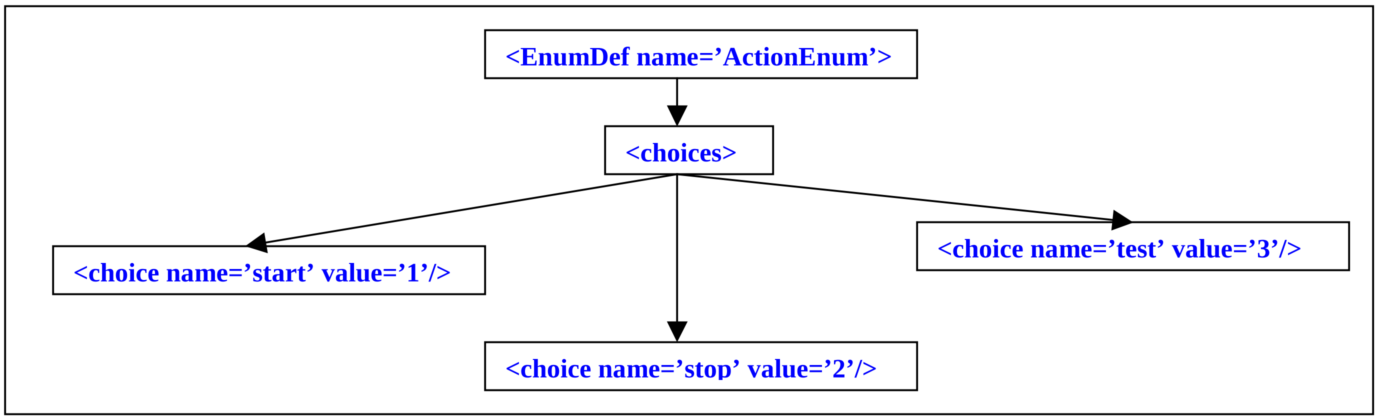

Figure 6: Part 2 of the tree structure

For the sake of accuracy, it should be emphasized here that the actual tree structure is much more detailed than what is depicted here. For example, attributes are nodes; text values are nodes and so on. However, Figures 5 and 6 are sufficient to continue our present discussion.

The third step is to use XSLT and XPATH facilities to process this tree to produce the output files. The XSLT script processes the **`AllEnums.xml`** file and produces one output file named **`enum_codegen.snp`** which is processed (snipped) by a utility program (file snipper) to produce four java files named **`AccessEnum.java`**, **`SeverityEnum.java`**, **`ActionEnum.java`** and **`AccessControlEnum.java`** respectively. For the sake of simplicity, each Java class has only one method whose purpose is to return the string value of the enum, given the integral value. After setting the classpath environment variable to contain **`saxon6.4.3.jar`**, the following command will execute the XSLT script to generate code for the enumerations.

```
java com.icl.saxon.StyleSheet –o gen_enum.snp AllEnums.xml
enum_codegen.xslt
```

What this command is saying is ‘execute the XSLT processor `com.icl.saxon.StyleSheet` to produce output file **`gen_enum.snp`** from input file **`AllEnums.xml`** by using XSLT script file **`enum_codegen.xslt`**.’

In Figure 7, the XSLT script and the output it produces are shown side by side. The output is shortened for brevity.

Enum_codegen.xslt(version 1)

```
<?xml version=’1.0’?>
<xsl:stylesheet version=’1.0’
xmlns:xsl=’http://www.w3.org/1999/XSL/Transform’>
<xsl:output omit-xml-declaration=’yes’/>

<xsl:template match = ‘/’>
<xsl:for-each select=’//EnumDef’’>
//@@@BEGIN_FILE <xsl:value-of select=’@name’/>.java
//@@@LOCATION common.gencode.enums
//****************************************
//*********** Generated code.  ************
//****************************************
public class <xsl:value-of select=’@name’/>
{
   public static String getEnumValueAsString(int
enumValue)
   {
   }
}
//@@@END_FILE AccessEnum.java
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

gen_enum.snp

```
//@@@BEGIN_FILE ActionEnum.java
//@@@LOCATION common.gencode.enums
//****************************************
//*********** Generated code.  ************
//****************************************
public class ActionEnum
{
   public static String getEnumValueAsString(int enumValue)
   {
   }
}
//@@@END_FILE ActionEnum.java
…….
Repetition of above pattern for SeverityEnum.java,
AccessEnum.java, AccessControlEnum.java
…….
```

Figure 7: The XSLT script and the output produced

The code generation **`script enum_codegen.xslt`**, version 1, could be explained as follows:

1. The first four lines are declarations, which are not very relevant from a code generation point of view.

2. The line **`<xsl:template match = '/'>`** instructs the XSLT processor to find the root node in the tree and apply the rules contained in the template body. The body works as follows:

    a. The `xsl:for-each` loop selects **`<EnumDef>`** child nodes within the root node or from its descendants. There are four **`<EnumDef>`** children found within the node hierarchy starting from the root node. So, the `xsl:for-each` loop will execute four times. Please refer to figure 5.

    b. For each execution of the `xsl:for-each` loop, all non xsl text is copied to the output. That means the first line within the `xsl:for-each` loop copies **`//@@@BEGIN_FILE`** to the output. The XSLT processor then processes the instruction **`<xsl:value-of select='@name'/>.java`** which outputs the value of the **name** attribute of the current **`<Enumdef>`** child followed by the **.java** extension. In a similar manner, you can analyze what happens for the other lines within the `xsl:for-each` loop.

    This step illustrates why it is important to visualize the input tree. At every instruction, we are using the XSLT or XPATH facility to navigate the input tree and select content from it to mix with our java bits and pieces to produce the output java files. Without a clear idea of the input tree, it would not be possible to use XSLT effectively to generate the desired code.

Hopefully, this explains the code generation development process with XML/XSLT. We are now in a position to show the XSLT script fragment, which completes **`getEnumValueAsString()`.** Please refer to Figure 6.

XSLT code fragment	**Corresponding generated code fragment**
Public static String getEnumValueAsString(int	**public static String getEnumValueAsString(int**
enumValue)	**enumValue)**
{	**{**
switch(enumValue)	**switch(enumValue)**
{<xsl:for-each select='choices/choice'>	**{**
case <xsl:value-of select='@value'/>:	**case 1:**
return "<xsl:value-of select='@name'/>;	**return "start";**
</xsl:for-each>	**case 2:**
	return "stop";
default:	**case 3:**
return null;	**return "test";**
}	
}	**default:**
	return null;
	}
	}

Figure 8: Completing the code generation for method getEnumValueAsString

Generative approach in an Application Software Development Project

The purpose of the project was to develop a Network Management System (NMS). Network management systems are like typical IT applications with the exception that they have to manage the information of a communication network. As a result, a major portion of the NMS igoes into providing network device access through some management protocol. Another core part of the NMS is the object infrastructure, which captures the information of the managed network based on an object and relationship model. The project benefited heavily by using a generative approach in these core areas. As was illustrated earlier, a lot of Java enumeration code was also generated. Although the project did not undertake, it was possible to generate other kinds of documents from the XML data the project had.

Network Access code generation

Simple Network Management Protocol is a UDP based network management protocol applied extensively in the data communication industry. Data exchanged over SNMP is defined in Management Information Base (MIB) with a language called Abstract Syntax Notation One (ASN.1). A commercial ASN.1 MIB compiler was used to 'dump' the MIB in text form. The MIB dump was then processed by a Java utility program to convert the ASCII MIB dump to XML. Essentially, the XML form has every bit of information that is available in the ASN.1 based management information base.

Once the SNMP MIB was available in XML form, XSLT processing was applied to generate object-oriented Java APIs on a very high level (i.e far removed from the drudgery of programming according to low level SNMP APIs). The high level SNMP APIs were flawlessly used by all project software developers. This approach could be compared with CORBA. Before CORBA, distributed application programming used to be done by expert TCP/IP programmers. With the advent of CORBA, TCP/IP based code was generated from the contract language. Thus, distributed application programming nowadays no longer requires TCP/IP experts. With SNMP code generation in our project, device control code no longer required SNMP experts.

There is another benefit of generating high-level APIs for network access. Two flavors of implementation were generated. One flavor provides network access by way of SNMP. Another flavor simulates network access by storing/retrieving data from local files. Application code using the high level API remains unaffected when one implementation is switched with another. Code generation for file based SNMP simulation allowed the project to proceed without waiting for the actual device to be ready. This is a tremendous advantage since network management development can proceed in parallel and can be tested with a very large simulated network.

Server side code generation

An object based server side infrastructure was used in this project. Application servers complying to the Enterprise Java Bean (EJB) standard provide concurrency, transaction, security, persistence, and naming services for the objects. A server side system development consists of implementing the information model by using Enterprise Java Beans and providing interfaces for remote access to the information which satisfy graphical user interface use cases. The project specified the information model along with a number of relationships in XML. A fictitious object showing object and relationship model capability is shown below. The model specifies the following:

- The Employee object has attributes like salary, job-title, join-date. It will inherit other attributes from the Person object.
- The Employee object cannot exist without a containing Division object.
- If one has access to the Employee object, one can get access to all its subordinates, which are objects of class Employee.
- If one has access to the Employee object, one can get access to the supervisor, which is an object of class Employee.

```
<managed-object class-name = “Employee”>
<base-object class-name = “Person”/>
<containing-object class-name = “Division”/>
< one-to-many-relation  role-name = “my-subordinates” class-name =
“Employee”/>
<one-to-one-relation  role-name = “my-boss” class-name =
“Employee”/>
<attribute name = “salary” type = “float”/>
<attribute name = “job-title” type = “string”/>
<attribute name = “join-date” type = “java.util.Date”/>
</managed-object-class>
```

Figure 9: An example of object and relational modeling in XML

All of the above specifications were implemented by auto generation. Similarly, a relational database schema in SQL was generated for implementing persistence of object and relational model information. XML deployment descriptors were generated for runtime deployment of Enterprise Java Beans. Further, semantics were imparted by using application classes inheriting generated classes, and application specific behavior was coded in derived classes.

Object and relational model code generation coupled with object services provided by the EJB framework created a very powerful paradigm of server side infrastructure development for the project. The project had a tremendous lead by being able to build further on this sophisticated server side infrastructure rather than spending time on building the infrastructure itself. The project was totally focused on building application logic and delivering functionality.

Limitations

The approach of code generation starting from XML and applying XSLT has some limitations. We experienced these limitations when doing extensive code generation work in real application software projects. These limitations are listed below.

- The input language is XML. XML syntax may be abhorrent to some. XML should be considered an underlying data representation language. Tools such as editors, both textual and visual, provide high level browsing and editing capabilities. These tools should be used while working with XML. If XML is generated from some other source (like ASN.1 mib), then this concern does not arise.

- XSLT is a purely functional language, which means there are no side effects allowed. For example, to implement a for-loop of fixed count one needs to implement a tail recursive template with a suitable termination condition. There are many more 'habit adjustments' application programmers need to make before becoming comfortable with XSLT programming. Moreover, XSLT has a 'logic programming flavor', a paradigm with which some application developers may not be familiar. XSLT syntax is quite verbose which is bothersome to say the least. XML escape mechanisms needs to be used for generating symbols like '<' (less than), '>' (greater than) and others which are heavily used in source code. Some XSLT workbenches are available to mitigate this inconvenience.

- There is no Integrated Development Environment (IDE) available for XSLT yet. Unlike the IDEs we have for C++/Java development with graphical debugging capabilities, no such things exist yet for XSLT . The debugging technique that was followed is to print part of the tree in the output document itself, i.e., by using <xsl:value-of ..>. The printed tree was then inspected to understand how a desired selection/transformation could be caused using XSLT/XPATH.

- The XSLT processor is quite permissive with regards to mistakes/omissions made in input the document. For example, missing elements or attributes do not cause the XSLT processor to fail. They cause missing output! Consequently, for source code generation, there will be compiler errors somewhere down the line. There are a number of solutions to this problem of input document validation. For example, one can use Schematron, an XSLT based assertion facility for XML document validation. XML schema enabled parsers may be able to do some validation based on the schema description of the document.

- XSLT is not a complete programming language. It is not possible to do certain kinds of transformations using just XSLT. There are a number of efforts in achieving XML transformation parallel to XSLT. Take a look at ***fxt***

(Functional XML Transformation) by Alexandru Berla et al and ***GSLgen*** from iMatrix.

Regardless of the above limitations, it could be argued that XSLT is being used to support application software development rather than being used directly in application software development. Consequently, project members' exposure to XSLT will be minimal once the correct code generation has been figured out. For most code generation purposes, an XSLT based solution is feasible. It is our belief that an XML based document transformation approach needs to become more widely known among application software developers in order to achieve its benefits as demonstrated in this paper.

Conclusion

This paper showed how XML based document transformation technology can benefit application software development projects. Here are some software metrics to show you how our project benefited:

- Out of approximately 2300 java files in the project, 1900 files were generated.
- SNMP coding was extremely easy. For example it took only one line of Java code to display the RFC1213 ipRouteTable as shown below

System.out.println(new IpRouteTableUtils().getIpRouteTable(deviceInfo));

`IpRouteTableUtils` is generated from RFC1213 ASN.1 MIB. The method **`getIpRouteTable`** returns the table object which is displayed in the **`System.out.println`** call. The **`deviceInfo`** object has the device IP address and the SNMP read community.

With the standardization of XML transformation by XSLT/XPATH, and with the increasing availability of good literature on the subject, it is our firm belief that code generation by XML transformation using XSLT will give application software development projects tremendous leverage in attaining increased productivity without sacrificing quality. One developer's expertise in XSLT and code generation could dramatically lessen the burden on other software developers by providing semantically rich, high-level APIs through code generation. With XSLT 2.0 features, the task of code generation will become easier.

Author Biographies

Soumen Sarkar is currently Lead Software Engineer in Atoga Systems Inc. He specializes in building large-scale distributed object systems, network management protocols and software architectures. He has more than ten years of experience in building object oriented software.

Craig Cleveland is an independent software consultant, instructor, and author of “Program Generators with XML and Java”. He specializes in domain engineering, Internet applications using Java and XML, and software architectures. Previously, he worked at AT&T Bell Labs developing and promoting program generator technologies. At Internet Games Corporation, Craig designed and implemented multi-player game sites including rating systems and fully automated tournaments. Visit him at http://craigc.com.

References

1. Program Generators with XML and Java by J. CRAIG CLEAVELAND.
2. XSLT 2nd Edition, Programmer’s Reference by Michael Kay.
3. SAXON XSLT Processor, **http://saxon.sourceforge.net/**
4. GSLgen, **http://www.imatix.com/html/gslgen/index.htm**
5. Fxt, generator of XML document transformers, **http://www.informatik.uni-trier.de/~aberlea/Fxt/**
6. Schematron: A XML Structure Validation Language, **http://www.ascc.net/xml/resource/schematron/schematron.html**
7. XML in 10 points, **http://www.w3.org/XML/1999/XML-in-10-points**
8. Extensible Markup Language (1.0), **http://www.w3.org/TR/REC-xml**
9. XSL Transformation (XSLT) Version 1.0, **http://www.w3.org/TR/xslt**
10. XML Path Language (XPATH) Version 1.0, **http://www.w3.org/TR/xpath**