XML Rule Based Source Code Generator for UML CASE Tool

Dong Hyuk Park, Soo Dong Kim
Department of computer Science
Soongsil University
Sangdo-dong, Dongjak-Ku, Seoul, Korea
E-mail: dhpark@selab.soongsil.ac.kr, sdkim@computing.soongsil.ac.kr

Abstract

Generating program source code based on design model by using CASE tool is one of the important areas in forward engineering. The generation of code from design model is valuable in making developers maintain consistency between a model and its implementation and abating the routine work of writing skeleton source codes. But, implementing code generation in CASE tool is not simple due to various metadata format, language, and policies of adopting modeler's option. And because of the continuous introduction of development environment like EJB and COM, the extensibility of CASE tool becomes principal comparison point. We believe that it be a feasible solution to generating source code in various language based on generation rule.

In this paper, we propose XML based code generation rule and code generator. The proposed rule provides higher level constructs to the developer for describing the way of code generation. And by making the code generator independent of repository format, the increase of the applicability of the code generator is shown.

1. Introduction

Using CASE tool, generating program source code based on design model is one of the important area in forward engineering. The generation of code from design models is valuable in making developers maintain consistency between a model and its implementation and abating the routine work of writing skeleton source codes. But, implementing code generation in CASE tool is not simple due to various metadata format, language, and policies of adopting modeler's option. And because of the continuous introduction of development environment like EJB and COM, we expect that the extensibility of CASE tool becomes principal comparison point.

However, the study about the extensible source code generator is not taken enough. The existing solutions did not show the satisfactory output.

We believe that there are two branches in techniques that imbue code generator with the extensibility. The one is by descriptor prescribing code generation rule and the other is by replacing code generation module based on component architecture.

Because component architecture based technique is dependent on a specific OS platform, the range of the application is limited. Therefore, we choose rule based technique.

The rest of this paper is structured as follows. In section 2, we describe the related work. In section 3, we describe the definition of code generation rule and the design of code generator for UML CASE tool. In this paper, we propose XML based code generation rule and code generator

2. Related Works

Code generator is a major component of CASE tool in supporting forward engineering. It reads repository data and outputs source code in various kinds of programming languages. In this section, we introspects several code generator included in some commercial CASE tools.

Reports and code generation can be performed in MetaEdit+[17] with the Report Browser. This is a tool for accessing information in the repository and checking it, producing various reports, generating program code. In this tool, the mapping rule between model information in the repository and source code to be printed are described in text-based script language.

In Rational Rose[18] , the thing corresponding to code generator does not explicitly exist. Instead, using COM[19] component model, code generation components are contained in the tool as a plug-in. And this tool is able to add the support of new language in code generation dynamically because of being based on COM component model.

3. Design of Code Generator for UML CASE tool

0-7695-1083-3/01 $17.00 © 2001 IEEE

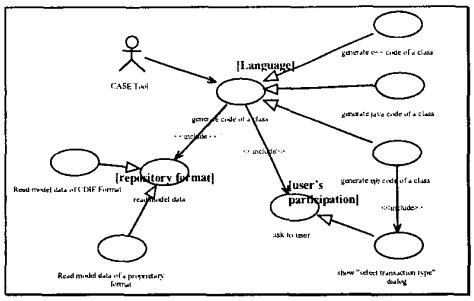

Figure 1. Variability focused use-case model for code generation.

Figure 1 shows variability focused use-case model for code generation. In the figure, three use-case variabilities are depicted on use-case model. By designating variability on use-case model, developer can be focusing on the variability in developing application. *Repository format* variability means that the variability in repository format exists between the two derived use-case of *read model data* use-case. *User's participation* variability means that there is the variability in the method for user to participate in process of code generation. *Language* variability means that CASE tool user requires the code generation of a various kind of language.

In designing the code generator, we should determine how to reflect the variabilities on implementation using what techniques. Our approach is to design adaptor classes for realizing *repository format* variability and to contrive a mapping rule descriptor for realizing user option and language variabilities.

Repository format variability appears due to intending to accept model data of various kinds of repository formats. To make our framework accept model data of various repository formats, we design *Model Data Extractor*, which is APIs designed for extracting model data from repository. To read model data from a repository format, one should designs adaptor classes for the specific repository format that implement the interfaces in *Model Data Extractor*.

3.1. Design Model Extractor

To date, various repository formats for CASE tool are available. Although the introduction of XML alleviates the incompatibility, because standard scheme is not prevalent and proprietary formats are still predominant, it is difficult to make our code generator understand such various formats. And even if those formats can be transformed into XMI, it is inefficient to include such a translation in code generation process.

In this section, we define an API for extracting necessary design model data from various repository formats. The non-functional requirement imposed on such an API includes the followings.

First, the API should be intuitive to API user. In this paper, we propose only API for extracting design model. Because the developer of data model extractor should implement logic for extracting design model data from a specific repository format, each operation and class of API should be not ambiguous to make easy to implement extracting logic. Second, the API should be independent of repository formats. By making API independent of repository format, we broaden the scope of application of our framework.

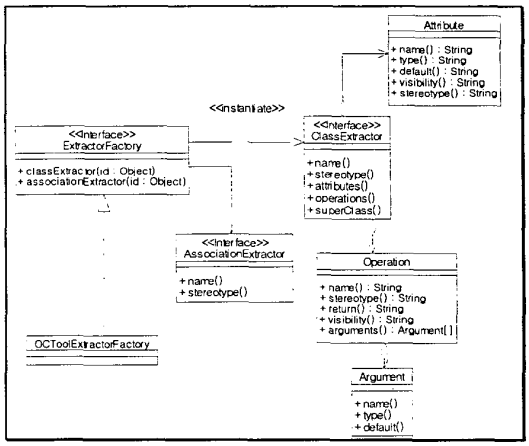

Figure 2. API for extracting class information

Figure 2 shows the API for extracting class information. The extractor consists of three parts, *ExtractorFactory*, *ClassExtractor*, and *AssociationExtractor*. *ExtractorFactory* is responsible for constructing the other extractor objects in the overall implementation. The user of our code generator must implements logic of constructing other extractor objects by creating a subclass of *ExtractorFactory* class like *CDIFExtractorFactory* class in Figure 2. In the figure, *CDIFExtractorFactory* class has logic of reading a repository file of CDIF repository format, extracting information related to requested model

ClassExtractor extracts information of a class in class diagram, which is denoted by user of CASE tool, and provides the information to CodeGenerator. *ClassExtractor* object is constructed by *ExtractorFactory*, which does such a construction in call of classExtractor(id). Id is an identifier object for designating a certain design model item. This object should be generic type. For example, in case of C++, this type may be void& or void* type and in java, may be Object type. By defining type of identifier as generic type, our framework can be repository format independent.

3.2. Mapping Design Model to Source Code

The next step of code generation is to apply a mapping rule to the extracted model data. By applying different

rules to the model data, user can generate source codes in various kinds of programming language. In this section, we introduce mapping rule descriptor and corresponding rule interpreter. Mapping rule descriptor is a document that describes how to translate model data to source code. Rule interpreter reads a mapping rule descriptor and then constructs code generator object and triggers code generation.

3.2.1. Mapping rule descriptor

Mapping rule descriptor is a XML document that describes how to translate model data to source code. This document consists of one header and several generations. Header part contains self-descriptive contents like as document name, used language, author and so forth. Each generation part describes rule for generating one file. Therefore, the number of generation part equals to the number of generated files.

Generation part contains relationship between model data and source code and consists of one organization and several structural elements. Structural element includes rules for generating source code like class, operation, and attributes. Structural element consists of class generation, operation generation, and attribute generation according to responsibility in generation process. Organization element defines the structure of code to be generated using other structural elements.

Each structural element represents code generation rule using *primitive* elements. Such primitive elements provide a kind of language for code generation to structural elements. Primitive elements include "Insertion", "Switch", and "Selection".

Table 1 provides the definitions of primitive elements and syntax. *Insertion* element is used to simply insert value of *text* attribute. In primitive elements, there are several attributes that are equally used. Such attributes are *text*, *place*, and *variable*. First, *text* denotes text string printed on source code. Second, place indicates where text string in *text* attribute should be output. Third, *variable* is utilized in conditional element to represent a variable operand in comparison. And to refer to model data in descriptor, text string prefixed with '$' in attribute value, which means model data in repository, is used.

Table 1. Primitive Elements

Primitive Element	Description	Syntax
Insertion	Insert string showed in text attribute	<Insertion place="string", text=string />
Switch	Conditional Insertion	<Switch place="string" variable="string"> <Case if="string" text="string"/> <Default text="string"/> </Switch>
Selection	Allow user to select one of several alternatives	<Selection place="string"> <Option text="string"/> <Option text="string"/> .... </Selection>
CustomInput	Allow user to insert arbitrary text string	<CustomInput place="string" />

Organization describes how to structure source code, so the structure and meaning of this part is heavily dependent on the syntax of programming language. And it differs from other three structural elements in used primitive elements. To make clear, we call it *node* element. Table 2 describes the definition of node elements used in organization part. In the Table 2, *Place* represents a place on which primitive generator produces a particle of source code. Place element have name attribute, which is identifier used when code generator finds primitive generator having the identifier.

Table 2. Node elements

Node Element	Description	Syntax
Place	Place where a particle of code is generated.	<Place name="string"/>
OnExist	Only if a specified place is processed, considered	<OnExist name="string"> .... </OnExist>
Opener/Closer	Opener / Closer	<Opener/>, <Closer/>
Repeat	Repeat generation	<Repeat on="modeData" firstOccurence="" " delimiter="," lastOccurence="" "> ..... </Repeat>
Class, Operation, Attribute, Argument	Used to structure source code and can embrace other *Node* elements	

Figure 3 illustrates an example of organization part. Organization part is encapsulated with *<Organization>* tag. One can control in what order to produce source code. In case of the organization shown in Figure 3, attribute code and operation code are produced within class's opener and closer. And *Repeat* element encircles attribute element to represent that attribute can be repeatedly appeared.

```
<Organization>
    <Class>
        <Place name="visibility"/>
        <Place name="type-keyword"/>
        <Place name="classname" />
        <Place name="linefeed" />
        <Place name="superclass" />
        <OnExist name="superclass">
            <Insert text="$linefeed"/>
        </OnExist>
        <Repeat on="$interface" firstOccurence="implements"
            delimiter = "," lastOccurence="$linefeed" >
            <Place name="interface" />
        </Repeat>
        <Opener/>
        <Repeat on="Attribute" lastOccurence="$linefeed" >
            <Attribute>
                <Place name = "visibility"/>
                <Place name = "static"/>
                <Place name = "final"/>
                <Place name = "type"/>
                <Place name = "attName" />
                <Insert text=";" />
            </Attribute>
        </Repeat>
```

Figure 3. Organization rule example

Figure 4 illustrates class generation in java. Class generation part is encircled with *<class>* tag and can include all primitive elements in the tag. In the figure, *insertion* element describes setting visibility of class to "public". And to generate different keyword according to the stereotype of class, switch element is used. Other *insertion* elements are used to generate java code corresponding to classname, superclass, and interface.

```
<Rule name="java", language="java">
    <Generation filename="$classname">
        <Class>
            <Insertion place="visibility" text="public"/>
            <Switch place="type-keyword" variable="$stereotype">
                <Case if="interface"
                    text="interface"/>
                <Default text="class"/>
            </Switch>
            <Insertion place="classname" text="$classname" />
            <Insertion place="superclass"
                text="extends $baseclass" />
            </Insertion>
            <Insertion place="interface" text="$interface" />
            <Opener symbol="{" />
            <Closer symbol="}" />
        </Class>
    </Generation>
</Rule >
```

Figure 4. Class generation rule example

Figure 5 represents operation generation rule example. In the figure, operation generation rule is split into operation part and argument part. First, operation part starts with selecting visibility of an operation, outputs return type and operation name, and defines opener and closer symbol.

```
<Operation>
    <Switch place="visibility" variable="$op_visibility">
        <Case if="public"
            text="public"/>
        <Case if="protected"
            text="protected"/>
        <Case if="private"
            text="private"/>
        <Default text="public"/>
    </Switch>
    <Insertion place="return" text="$return" />
    <Insertion place="op_name" text="$op_name"/>
    <Opener symbol="{" />
    <Closer symbol="}" />
</Operation>
<Argument>
    <Insert place="type" text="$type" />
    <Insert place="variable" text="$variable"/>
</Argument>
```

Figure 5. Operation generation rule example

3.2.2. Rule Interpreter

It is the principal responsibility of the rule interpreter that constructs a virtual code generator based on mapping rule descriptor. It reads mapping rule descriptor and outputs virtual code generator, and then the produced code generator is used repeatedly in code generating several classes. By reusing constructed code generator and reading descriptor only once, one is able to reduce the time taken in generating source code.

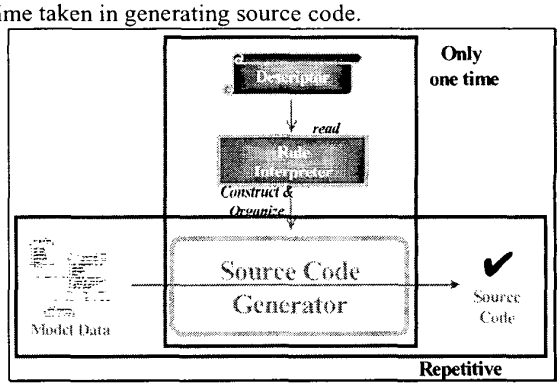

Figure 6. Overall Process of generating source code

Figure 7 illustrates the process of constructing code generator. The procedure of constructing code generator is split into two processes, constructing primitive generators, and organizing the primitive generators. Using mapping rule descriptor, rule interpreter constructs primitive generators corresponding to primitive elements and maintains the primitive generators using hashmap, in which we use *place* attribute of primitive elements as key and primitive generator as value. This map is used as a catalog by organizer for code generation. And then rule interpreter reads organization part of the descriptor and constructs generation tree based on structural elements.

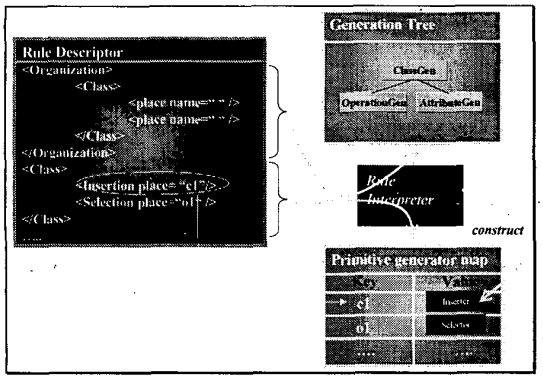

Figure 7. Process of constructing code generator

```
createPrimitiveGeneratorMap(StructuralElement se)        Called per each structural element
{
        elements = se.getAllPrimitiveElement();
        PrimitiveGeneratorMap pgm = new PrimitiveGeneratorMap(se)
        while (elements.hasNext() )
        {
                PrimitiveElement pe = elements.next();
                PrimitiveGenerator pg = createPrimitiveGenerator(pe);
                pgm.add(pg);                             Add PrimitiveGenerator to a map
        }
        return pgm;
}

createPrimitiveGenerator(PrimitiveElement pe)
{
        PrimitiveGeneratorFactory pgf = getFactory(pe);
        return pgf.create(pe);
}
```

Figure 8. Pseudo code of constructing primitive generator map

The construction of *Primitive generator map* is conducted in the following steps. *RuleInterpreter* accepts one structural element and introspects internals of it. During the introspection, RuleInterpreter accepts *primitive* elements, constructs an appropriate primitive generator object, and save the object to the map object for the structural element. Figure 8 shows the pseudo code illustrating how for primitive generator map to be created.

Interpreter class is divided into two classes according to the kind of the element to interpret. One is *NodeInterpreter* class and the other is *GeneratorInterpreter* class. *NodeInterpreter* derivations read *Organization* part and generate *Node* derivations. And *GeneratorInterpreter* derivations accept Generation part and output *PrimitiveGenerator* derivations. The generated *PrimitivedGenerator* derivations are saved into *PrimitiveGeneratorMap* object, which is used in generating source code by *Node* derivations.

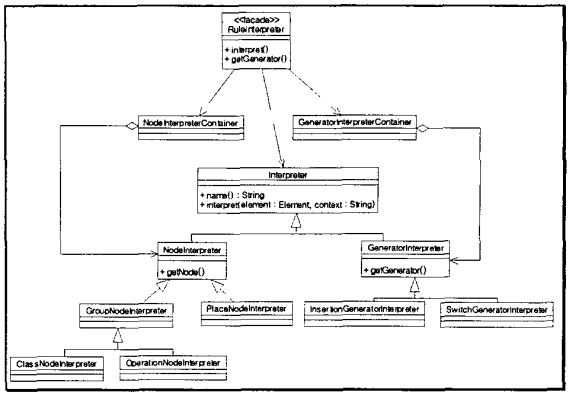

Figure 9. Class diagram of RuleInterpreter hierarchy

Each interpreter class is maintained in the corresponding container class and retrieved and requested to interpreter mapping rule descriptor by *RuleInterpreter* class. The rule interpretation proceeds through following steps and repeats until there are no elements in rule descriptor.

1. Obtains elements in rule descriptor.
2. Retrieve the appropriate node interpreter from container class.
3. Request to the node interpreter for creating a node.

4.a Register the node to the parent node. (in case of NodeInterpreter)

4.b Register the node to the PrimitiveGeneratorMap (in case of GeneratorInterpreter)

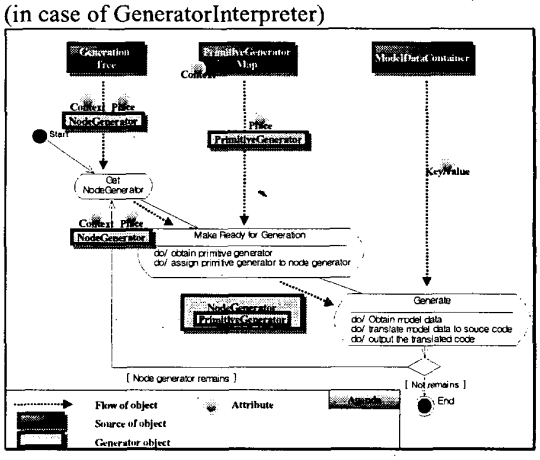

Figure 10. The code generation process of code generator

Figure 10 shows the process of code generation using generation tree and primitive generator map. The process consists of three parts. First, node generator is obtained from *generation tree*. And then node generator obtains appropriate *primitive generator* from primitive generator map. And then node generator collaborates with primitive generator to generate code with *model data container*.

3.2.3. Class Diagram and Sequence Diagram for code generator

Figure 11 illustrates class diagram of code generator. Class diagram for code generator is divided into rule interpreter, generation tree, and primitive generator.

RuleInterpreter class has the major responsibilities as the followings. The first is that it interprets mapping rule descriptor that is documented in XML. The second is that it instantiates primitive generator based on primitive elements shown on structural element. The third is that it constructs generation tree based on node elements described in organization part.

RuleInterpreter provides three APIs to support those functionalities. Function *interpret(filename : String)* takes a descriptor's filename as parameter and reads the specified descriptor, and then constructs and maintains generation tree, *PrimitiveGeneratorMap* and *CodeGenerator* object. Function *getPrimitiveGeneratorMap(context: String)* takes a context as parameter, in which context represents structural elements that encircles primitive elements in descriptor, and then returns *PrimitiveGeneratorMap* object having the designated context. Function *getGenerator()* just returns *CodeGenerator* object that is created when *interpret()* is called.

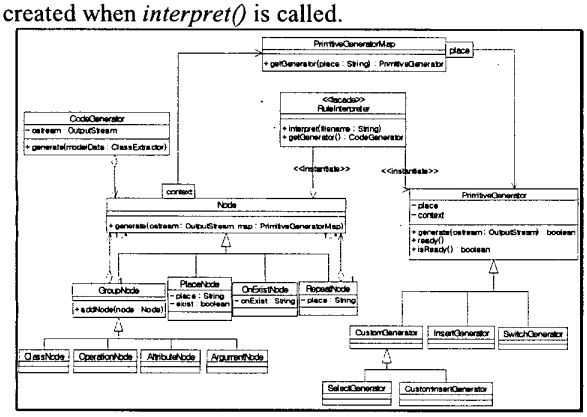

Figure 11. Class diagram of code generator

PrimitiveGenerator class is an abstraction of primitive elements in descriptor. Therefore, the derived classes of *PrimitiveGenerator* class contain concrete code generation logic. The main functionality of PrimitiveGenreator derived objects is to generate a particle of code responding to the request of generation tree. The derived classes of *PrimitiveGenerator* class include *InsertGenerator*, *SwitchGenerator*, *SelectGenerator*, and *CustomInsertGenerator*, among which *SelectGenrator* and *CustomInsertGenerator* are children of *CustomGenerator*. *InsertGenerator* class just generates string shown in *text* attribute, in which if $ prefixed string appears, generator obtains a string to be outputted from *ClassExtractor* or *reserved keyword map*.

SwitchGenerator class has the functionality similar to switch-case statement in programming language. This class maintains a list of condition and text pair. In response to the request of generation, it fetches variable data from *ClassExtractor* and then chooses appropriate text or default text by comparing variable and condition of each pair and output the selected text.

CustomGenerator is an abstraction of primitive elements that needs user's participation. In this context, user's participation means that during code generation, CASE tool requires the input of the user. *SelectGenerator* provides several options, among which a user selects one or several options. *CustomInsertGenerator* just generates string inputted by user.

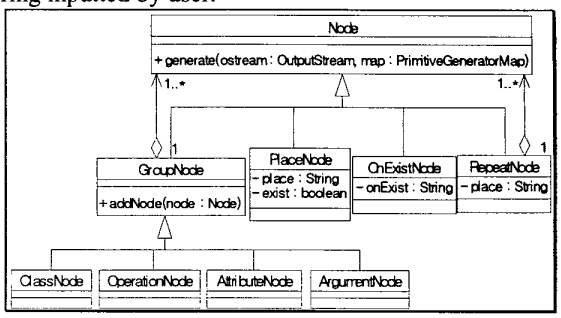

Figure 12. Inheritance Hierarchy of Node Generators

Figure 12 illustrates inheritance hierarchy of node elements. *Node* class and the derived classes of it are used to construct generation tree. The principal role of *Node* class and its derived classes is to delegate generation call to appropriate primitive generator object, in which process *Node* class uses *PrimitiveGeneratorMap* class to obtain the primitive generator object. *PlaceNode* class represents *Place* element appeared in descriptor. In the figure, *GoupNode* is able to aggregate other node objects, which is not reflected on descriptor and used to constructing generation tree by *RuleInterpreter*. *OnExistNode* class corresponds to the *OnExist* in descriptor. *OnExistNode* object finds *PlaceNode* object previously executed using *place* attribute and asks to the found *PlaceNode* object whether *PlaceNode* object is executed or not. And then if the *PlaceNode* object is executed, the *Node* objects contained in *OnExistNode* are executed. *RepeatNode* are used to generate a repeatedly appeared code. RepeatNode object investigates the *PlaceNode* object denoted by the place attribute and until the *PlaceNode* is not executed, the *Node* objects nested in *RepeatNode* object are executed. The classes derived from *GroupNode* class are used to organize the structure of source code to be generated and control the scope of the model data that the embedded *Node Generator* object can access.

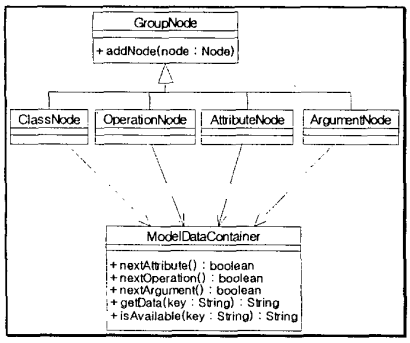

Figure 13. Class Diagram of ModelDataContainer

Figure 13 describes relationships among *ModelDataContainer* class and *GroupNode* derived classes. *ModelDataContainer* maintains model data having $ prefixed string as identifier and is utilized by all generator classes. Of several operations of *ModelDataContainer*, the operations starting with *next* have responsibilities a little different from other data extraction functions. Those functions, including nextAttribute(), nextOperation(), and nextArgument(), are used for accessing repeatedly appeared model data and called by *OperationNode*, *AttributeNode*, and *ArgumentNode*.

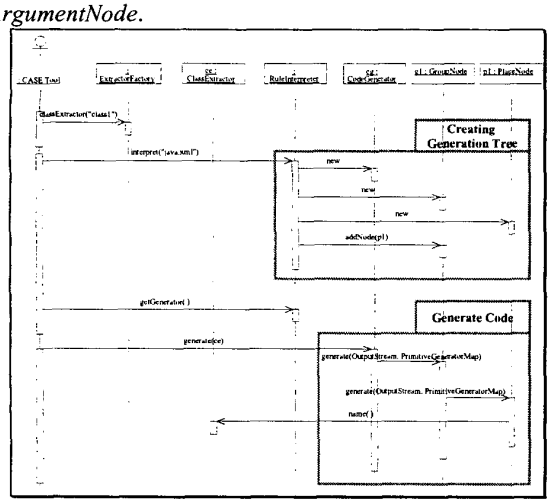

Figure 14. Sequence Diagram depicting Code Generation Process

Figure 14 shows message flows happened when CASE tool requests to generate a java code for a class. First, CASE tool obtains *ClassExtractor* object through *ExtractoryFactory* object by passing identifier of the class. And then it sends message "*interpret("java.xml")*" to *RuleInterpretor* to generate code generator, including the construction of generation tree and primitive generators. In the "*interpret*" method, *CodeGenerator* object is created, *Node* derived object is created and organized into generation tree, and primitive generators is created. And then CASE tool obtains *CodeGenerator* object from *RuleInterpretor* and sends message "*generate(ce)*" with *ClassExtractor* object as argument. The CodeGenertor, that receives "*generate*" message, sends message "*generate*" to generation tree. The *Node* derived object contained in the generation tree sends "*generate()*" message to the primitive generator of the node object and child *Node* object.

4. Assessment and Conclusion

4.1. Assessment

The code generator proposed in this paper has the following merits. It is able to add the functionality of code generation for new language dynamically. That is, to recompile and re-execute the application is not needed to support code generation in a new language. And by changing organization of rule descriptor, developer can control the structure of generated source code. And by generating several files using one descriptor, developer can increase coherence between related generation rules. We split the mapping rule descriptor into Organization part and Generation part. Due to the seperation, the complexity of each element in descriptor lessens.

However, the proposed solution is the lack of reusing mechanism, like inheritance, of predefined mapping rule. As a result, when describing rule descriptors for two homogeneous languages, developer should duplicate a lot of descriptor. And by distributing mapping rule to two parts, organization and generation part, while flexibility of mapping rule increases, complexity of rule becomes worse.

4.2. Conclusion

The generation of code from design models is valuable in making developers maintain consistency between a model and its implementation and abating the routine work of writing skeleton source codes. But, implementing code generation in CASE tool is not simple due to various metadata format, language, and policies of adopting modeler's option. And because of the continuous introduction of development environment like EJB and COM, the extensibility of CASE tool becomes principal comparison point.

In this paper, we propose the code generator and mapping rule descriptor to define the relationship between UML class and various kinds of programming source code. By developing code generation part of CASE tool using the proposed solution, the cost and time required is reduced. And because of being able to rapidly react to introduction of new language, the user of our solution can increase the market share.

Further work will be concerned with the support to other UML model, currently confined to class mode, and the reuse of predefined mapping rule.

5. References

[1] T. Lewis et al., Object Oriented Application Frameworks, Ed. Manning, USA, 1995.

[2] D. D' Souza, A. Wills, Objects, Components and Frameworks with UML – The Catalysis Approach, Addison-Wesley Publishing Company, 1999.

[3] Ivar Jacobson et al, Software Reuse – Architecture, Process and Organization for Business Success, Addison-Wesley, 1999.

[4] Kang K, et al, Feature-Oriented Domain Analysis (FODA). Feasability Study. Technical Report, CMU/SEI-90-TR-21, November. Software Engineering Institute, Pittsburgh, PA 15213, 1990.

[5] Itana M. S. Gimenes et al, An Object Oriented Framework for Task Scheduling, *In Proceedings of the 33th Technology of Object-Oriented Languages and Systems* (St. Malo, France), 2000

[6] Douglas Scmidt et al, Pattern-Oriented Software Architecture Volume 2 Patterns for Concurrent and Networked Objects, John wiley & Sons, 2000.

[7] David M. Weiss, *Commonality Analysis : A Systematic Process for Defining Families*, Second International Workshop on Development and Evolution of Software Architectures for Product Families, February 1998

[8] James O. Coplien, *Multi-Paradigm DESIGN for C++*, Addison Wesley, 1995

[9] Johnson, R.E. and Foote, B. Designing reusable classes, *J. Object-Oriented Programming 1,5 (June/July 1988)*, 22-35.

[10] Fayad, M.E., Schmidt, D.C., and Johnson, R.E, Object-Oriented Application Freameworks: Problems and Perspectives. Wiley, NY, 1997.

[11] Pree, W. Design Patterns for Object-Oriented Software Development, Addison-Wesley, Reading, Mass. 1994.

[12] Philippe Kruchten, Architectural Blueprints - The 4 + 1 View Model of Software Architecture, IEEE Software November 1995.

[13] Clemens Szyperski, *Component Software*, Addison Wesley, 1998.

[14] Capt Gary Haines, David Carney, John Foreman, *Component-Based Software Development / COTS Integration*, CMU Software Technology Review, October 1997.

[15] Fayad M.E., Schmidt D.C., *Object-Oriented Application Framework*, Communication of ACM, October , 1997.

[16] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Language Reference Manual, Addison-Wesley Publishing Company, 1999.

[17] Aonix, MetaEdit+, http://www.metacase.com/mep.

[18] Rational Software, Rational Rose, http://www.rational.com/

[19] Microsoft, The Component Object Model: A Technical Overview, http://msdn.microsoft.com/library/techart/msdn_comppr.htm, October, 1994.