VERSITA
Applied Computer Systems
doi: 10.2478/acss-2013-0002
2013/ 14

Code Generation from UML Model: State of the Art and Practical Implications

Andrejs Bajovs[1], Oksana Nikiforova[2], Janis Sejans[3], [1-3]*Riga Technical University*

***Abstract* – The paper draws attention to the problem of code generation under advanced software development. In their previous publications the authors already discussed several issues associated with the inconsistency between the generated software components and those expected in respect with the rules for object-oriented programming. The current research is an effort to systematize the information about code generation methods and techniques applied in the task of code generation and to try to answer the question of what is the reason for code generation failing to work correctly.**

***Keywords* – code generation, UML class diagram, code generation tool.**

I. Introduction

Despite high levels of IT technology and IT fields as such, development related problems that were traditionally associated with the development of software still have not been completely solved. Of course, one would always like to decrease the development time and costs, however, it must be understood that still too much time is being devoted to routine work. Therefore, reusability remains an important theme, because technology is changing much faster than business processes of the problem domain. The first reason, why this problem has not yet been solved is the technological diversity, which, in its turn, leads to such situations where every system is overwritten over and over again in a case of any change in technological architecture. Second, even if the problem domain is described with the help of "traditional" CASE tools, this does not solve the problem because created models are nothing more than documentation, so that they are not automatable to be transformed in executable program code.

Even more, further requirements are implemented directly in the code bypassing system documentation, thus no longer meeting the actual functionality. Hence, the fundamental problem of software development is a "semantic gap" between models and programming language, which permits transformation of the problem domain description into the executable code.

Human brain perceives graphical information much better than textual one that is why it is useful to display a system as a model during its creation phase. To accomplish this, the Unified Modeling Language (UML) [1] which is used to model object-oriented systems was developed. UML was created not only as a system specification tool, it is positioned also as a tool, which will allow to automatically generate the code from UML models. With such a position Object Management Group (OMG) announced its new invention, Model Driven Architecture (MDA), at the end of 2001. Since that time already 10 years passed, and 15 years elapsed since UML was standardized. During that time, a lot of different CASE tools have been developed, which are advertized as more or less capable of generating the program code from the system model. These are both open source tools and the environment as well as commercial products. Nowadays, MDA tools announced that they support translation of solution elements into software components, e.g. transformation from platform independent to platform specific model, and code generation. However nobody heard about a software system as yet that could be actually designed based only on the principles of MDA. The alternative to MDA and UML approach is Domain Specific Language (DSL), which allows specifying system behavior. However, DSL do not fall within the scope of this article. '.

Moreover, the results of the authors' previous research on code generation abilities in advanced modeling tools were quite discouraging [2]. The experimental model was created to check the ability of the tools to generate expecting program code from them. The experimental models contained more than 60 tests. The transformation results (generated construction) were classified into compilation errors, execution errors, information losses, missing notations, etc.

In many tests the investigated tools (Sparx Enterprise Architect, Visual Paradigm and Microsoft Visual Studio 2010) showed different results. And it was not like one tool always demonstrates better result than the other. There were tests, which showed the correct result, while other failed, and vice versa. The analysis of the transformation result shows that the quality of the generated source code is very low (in our experiments). Only 14% (in Sparx Enterprise Architect), 18% (in Visual Paradigm) and 18% (in Visual Studio) of the tests proved to be correct. Considering that these tests did not include all possible elements of the UML class diagram, such as various relationship elements, the obtained results may be still considered as one of the main reasons why the class diagrams and code generation from models are not so widely applied in the IT industry.

Therefore, the authors made code generation the object of their research and tried to investigate, why despite of the modeling standards containing descriptions of efficient model creation techniques there is still no clue of how to automatically transform a model of a complex system into a source code.

The goal of the paper is to research the main principles and approaches of the model to code transformations. The paper describes Model Driven Software Development principles and code generation related standards, which were created by

Object Management Group organization. The importance of code generation was substantiated. Because of the importance of this process, the related techniques, methods and approaches were studied more thoroughly, in addition, the code generators taxonomy was created. It was then used to classify modeling tools built in code generators. The paper also shows the qualitative comparison of the tools with the help of certain comparison criteria. Finally, the description of the tool created by one of the paper authors in the course of development of his bachelor's thesis [3] is presented. The program realizes a combination of template-based and frame-based code generation approaches.

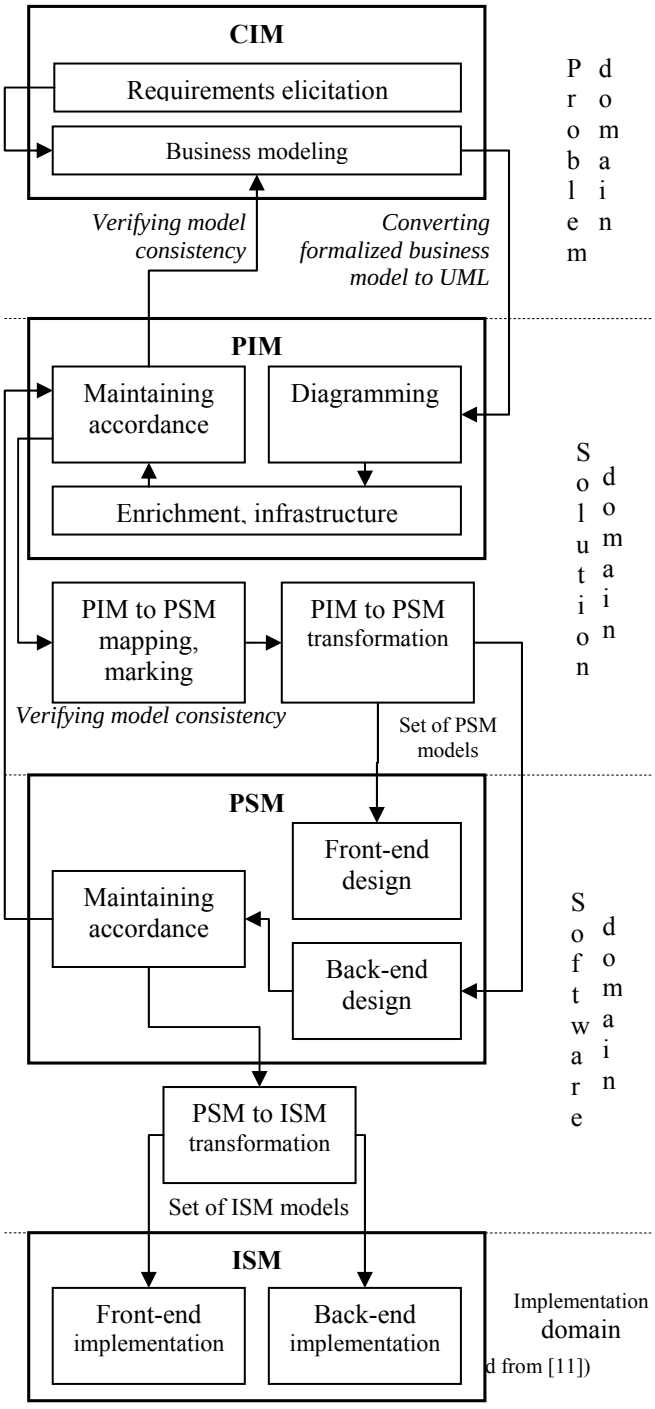
II. Principles of Code Generation under the Framework of MDSD and MDA

This paper owes its existence to the existence of the process called modeling. OMG organization gives the following definition of a model [4]: "*A model of a system is a description or specification of that system and its environment for some certain purpose.*" The framework called Model Driven Software Development (MDSD) or Model Driven Development (MDD) is very closely related to modeling. The importance of its usage in modern software development is proved by the words of the research area guru Markus Völter. He says: "Based on years of my own experience, as well as the experience of the overall MDD community, I am convinced that Model Driven Development is a step in the right direction and brings about significant improvements in productivity, quality, and overall system consistence."[5]

According to [6] model in MDSD is compact and formal at the same time. It has the exact meaning of the program code so that the model is not only documentation, but part of the software. MDSD appeared because there was the necessity to decrease efforts, to create and use analysis and design models at each stage of the software development process and to automate transformation of the models [7]. MDSD also defines how to separate high-level business logic from system architecture and deployment platform. However, such term as Model Driven Architecture (MDA) is much commonly used nowadays when the talk is about usage of models in software development.

In general, MDSD and MDA are very similar terms [8, 9]. The Model Driven Architecture was introduced by OMG organization in 2001 as an effort to standardize the MDSD framework. Unlike MDSD, the MDA approach considers models to be central in the development process (assuming that the model represents a set of diagrams, used to express the whole software system) [10]. Generally speaking, MDSD is less strict and much blurry than MDA. That is why Model Driven Architecture may be considered as a next stage in the evolution of software development process, which tends to bring some improvements to each step of the software development life cycle.

The MDA proposes to construct four basic models for the developed system (see Figure 1).



The four basic models for the developed system are:

1. Computation Independent Model (CIM) that reflects business and its models — defined at the problem domain level in Figure 1;

2. Platform Independent Model (PIM) that reflects analysis and design models of the software system to be developed— defined at the solution domain level in Figure 1;

3. One or many Platform Specific Models (PSM) that reflect detailed design models of software system under construction—defined at the software domain level in Figure 1;

4. One or many Implementation Specific Models (ISM) that reflect implementation and runtime models – defined at the implementation domain level in Figure 1.

The components of MDA, shown on Figure 1, represent all of the activities included in the MDA-driven software development process. Dependence on information exchange, which is imported/exported from one component to another, is written on the arrows between the components in Figure 1. Regular font on the arrows between the components means that import/export of models is possible. Transitions between the components, which can be performed only by a human at the moment, are expressed in italics.

The process, which converts one model into another is called transformation. As shown in Figure 1, CIM to PIM transformations are usually performed manually. As for the PIM to PSM as well as PSM to ISM transformation, software developers are trying to perform it automatically. In the authors' opinion it is much easier to perform these operations in comparison to PSM to ISM transformation because in the first case similar model forms are used (graphical). In the second case graphical model is transformed directly into text. This paper mostly deals with PSM to ISM transformation, which is more commonly called 'code generation'.

III. Review of Code Generation Methods and Techniques

With the announcement of MDA Object Management Group also tried to describe how different parts of this architecture would work. While the OMG organization developed theoretical basis of the research area, the practical side of code generation started to fall behind. Nowadays a lot of different standards related to code generation exist [12], but no method is able to provide comprehensive guidelines of how to convert all this theory into practice. However, various methods and techniques do exist, which try to define certain important aspects of the code generation process.

A. Code generation approaches

According to [13], the two main code generation approaches can be singled out: visitor-based and template-based.

The point of the *visitor-based* approach is that the code is being generated while iterating through textual representation of the model.

The *template-based* approach is more commonly used because it could provide more complex and "nice" code generation. This approach implies writing special textual templates, which basically are a set of rules. These rules specify the way of generating code from some specific model. However, the most significant disadvantage of this approach is that with the growth of computer system, the template could quickly become too complex [5]. That is why it is more preferable to modify this approach, for systems that are processing a lot of data to accomplish different sets of tasks. It could also be wise to use such approach in combination with the first one.

B. Code generator's types

Five different types of code generators can be singled out [14]:

Code munger is the simplest of these. „Mung-ing" means transforming something from one form into another form, thus the generator goes through the input and searches for important features (keywords, tags, etc.) and uses them to create the output files. This kind of generator can be used to generate documentation, retrieve and collect some specific information or generate some kind of the input analysis result.

An *Inline-code expander* reads the source code and, where it finds the predefined markup, inserts the production code. The difference to code munger is that the output is the same as the input but with corrections - additions. The negative aspect is debugging, because markup words may cause compilation errors, and even if the markup is in the comment form, the source code does not represent full execution code.

Mixed-code generator is the combination of the previous two types. The difference is that the generator output can be used as the input (modifications are done in the input file), thus the result can be regenerated. Similar to the inline-code expander the generator reads the source code and finds the predefined markup but instead of inserting the production code the mixed-code generator replaces the area between the markup. Because the markup is specially formatted comments, which mark the start and the end of the replacement area, the source file can be used for debugging.

The *partial-class* generator input is the definition file and template files. By analyzing the definition file, the template is filled and the output is produced. The output of the generator will depend on the variable information in the definition file. The partial-class generator is used for generating base structures or base classes which afterwards are manually updated with final functionality.

Tier or layer generator is similar to partial-class generator, except it takes the responsibility to completely generate one tier of an n-tier application. This means, that the generated code has enough functionality for the working tier or layer, and not just base classes. Tier or layer generator can be positioned as a Model Driven generator, where the UML model is an input and one or more tiers of the system is an output.

C. Code generation techniques

The paper [5] describes such code generation techniques as:

Templates and filtering technique defines that generalized source code fragments are represented as textual templates. During code generation process, the variables of these templates are associated with the system model.

Templates and meta-model technique is very similar to the previous one, except the new meta-model of the system representation is being created before the template is executed in a user defined way. It means that templates are not dependent on the model syntax and could be written in any programming language.

Frame processors have the following principle of operation: the code is being specified with the help of object-

like frames. Each frame owns a group of attributes called slots. When a new frame is being created, its slots are connected with specific values. Thoey may represent different aspects of thye system model and also may contain references to other frames. Thus, during code generation, the frames form the tree structure of the system source code.

API-based generator is usually connected with a single programming language. Thus it provides the user with a special framework, which makes the code generation process more intuitive. The source code is commonly specified with the help of templates. API-based generators often have built-in compilers that are able to evaluate the generated code at once.

In-line generation technique determines that the program source code contains text fragments, which at the time of compilation, depending on the conditions, can be extended or not included into the program at all. Such code generation technique is quite primitive, therefore it cannot be used as the main one.

Code attributes technique defines that special text such as comments can be placed in the existing source-code. In fact, these comments are specific instructions to the code generator. This technique is commonly used to automatically generate program documentation.

Code weaving technique states the following: some independent and non-related pieces of text are manually written into the input file. At the same time it is also defined how all these fragments should be joined together during the code generation process. Finally, all these pieces would be processed and mixed to form the program source code.

D. Code generator taxonomy

The earlier defined code generation approaches, types and techniques can be connected as follows: code generators can be derived from one of the code generation approaches: visitor-based or template-based. They differ from specific generators types, which implement specific code generation techniques.

The essence of the template-based approach taxonomy is illustrated in Figure 2.

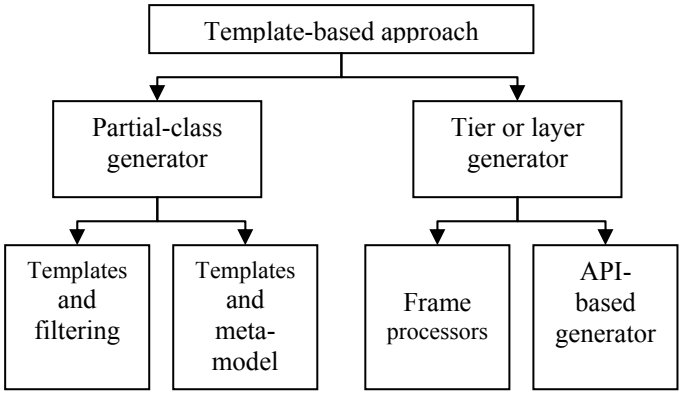


Fig. 2. Template-based approach taxonomy

Templates and filtering as well as templates and meta-model techniques are associated with the partial-class generator because it operates with templates. These techniques are also the simplest - they allow generating only the carcass of a system code. Tier or layer generators have a more complex mechanism that is why they implement frame processors and API-based generators.

The essence of the visitor-based approach taxonomy is illustrated in Figure 3.

Code attribute technique is associated with code munger type generators because of their ability to automatically create program documentation. In-line generator technique is implemented in the in-line code expander type generators because they do not change the form of input source code but add a new fragments to it. Code weaving technique has both code munger and in-line code expander characteristics: these code generators are capable of completely changing the input file or, just adding some new code fragments. Mixed code generator should not be fully associated with any of the techniques listed above. Code generator may or may not have any features of this type depending on its purpose and set of tasks.

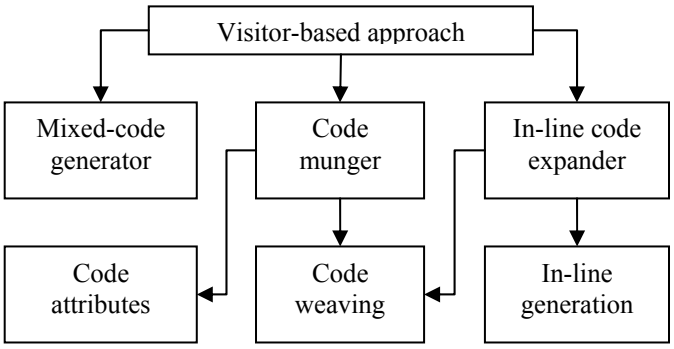


Fig. 3. Visitor-based approach taxonomy

The presented taxonomies are quite strict, however in practice one code generator can have both the features of the visitor and template-based approaches. Code generators are not isolated and differ from each other, yet complement each other. Because of that, it can be difficult to apply strict classification to a single code generator.

IV. Revealing the Real Nature of Existing Code Generators

The taxonomy of code generators presented in the previous section raises some questions: why the existing tools, which have some code generating functions, are not so widely used in the industry [15] when there are various methods of code generation? Can the great corporations like Sparx or IBM fully use all these methods to create really functional code generators?

A. Code generators' classification

To answer these questions, the authors attempted to classify some of the famous tools, which according to their developers, grant broad code generating functionality. The four tools participated in the experiment: IBM Rational Software

Architect 8.0, Sparx Enterprise Architect 9, Microsoft Visual Studio 2010 and Eclipse 3.7.2. These tools are quite popular and were developed by the famous and big companies (a huge group of people in case of Eclipse). It means that they should be quite functional and qualitative. In general, all of the chosen tools are slightly different in their functionality: Sparx Enterprise Architect 9 is well known as a UML modeling tool [16], IBM Rational Software Architect 8.0 is positioned as a huge project-developing framework [17]. Microsoft Visual Studio 2010 is a development environment and its modeling functionality is much smaller than previous ones [18]. Eclipse is an open-source plug-in based product, which theoretically could be extended to perform any types of tasks [19]. Basically, Eclipse does not have built-in code generation functions, but this can be easily fixed with such add-ons as Acceleo or XPand.

Since tool developers do not provide exact information about the mechanisms their code generators are using, the authors of this paper may only guess this by reading documentation and study their functionality. The tools were associated with concrete methods or types from the taxonomy only if authors were fully sure about this.

Table 1 shows tool classification in relation to code generation techniques.

TABLE I

TOOLS' CLASSIFICATION RELATIVELY TO THE CODE GENERATION TECHNIQUES

	Tools			
Code generation techniques	*IBM Rational Software Architect 8.0*	*Sparx Enterprise Architect 9*	*MS Visual Studio 2010*	*Eclipse 3.7.2*
Templates and filtering		X	X	X
Templates and meta-model				
Frame processors				X
API-based generator	X	X	X	
In-line generation	X	X	X	X
Code attributes	X	X	X	X
Code weaving				

IBM Rational Software Architect realizes the so-called comment templates, which are associated with the *code attribute* technique. The tool provides API to specify code generation process. It means that Rational Software Architect is an *API-based generator*. The *in-line code generation* technique may also be associated with this tool because it supports C++ programming language, which has the "define" instruction [20].

Sparx Enterprise Architect has the built-in language for writing templates [21]. That is why this tool may be associated with *templates and filtering* technique as well as *API-based code generation* technique. Sparx Enterprise Architect also supports C++ and some other programming languages, which are connected with the in-*line code generation* technique. The tool supports at least JavaDoc documentation, which means that Sparx Enterprise Architect could be associated with the *code attributes* technique [16].

Visual Studio 2010 gives an opportunity to define text templates by using C# code fragments [22]. This allows to associate Visual Studio 2010 with the *templates and filtering* as well as the *API-based generator* techniques. The *in-line code generation* technique can also be associated with this tool because of C++ programming language support. C# programming language has an ability to automatically generate documentation (XML documentation), which is the feature of the *code attributes* technique.

Eclipse also provides the developer with the opportunity to write templates, thus it realizes the *templates and filtering* code generation technique. Eclipse also has the so-called workflow code generation mechanism, which at its core is very similar to the principles of *frame processing* technique. The tool supports some programming languages such as C++, which allows associating it with the *in-line code generation* technique. The *code attributes technique* is also connected with Eclipse because of its ability to automatically generate program documentation, such as JavaDoc [23].

According to the classification as well as considering the taxonomy presented above, the chosen tools were also classified by types of code generators. The results of this are shown in Table 2.

TABLE II

TOOLS' CLASSIFICATION RELATIVELY TO THE CODE GENERATORS TYPES

	Tools			
Code generators types	*IBM Rational Software Architect 8.0*	*Sparx Enterprise Architect 9*	*MS Visual Studio 2010*	*Eclipse 3.7.2*
Code munger	X	X	X	X
In-line code expander	X	X	X	X
Mixed-code generator				

Partial-class generator		X	X	X
Tier or layer generator	X	X	X	X

The presented classification shows that some popular tools realize a lot of various code generation mechanisms and thus are focused on solving quite a wide range of problems. They realize both template-based and visitor-based code generation approaches. Therefore, it sounds quite logical to state that it is sufficient to make the mentioned tools work well,. However, in practice they are not very widely used.

B. Qualitative evaluation of code generators

Unpopularity of code generators may be connected to the fact they are too inconvenient to use or lack certain modern vital features. The evaluation of the code generators quality was made to prove this. The tools mentioned in the previous section (IBM Rational Software Architect 8.0, Sparx Enterprise Architect 9, Microsoft Visual Studio 2010 and Eclipse 3.7.2) were also tested for their quality..

First of all, some criteria were considered with the purpose to find out more about the quality of code generators. Table 3 shows these criteria and provides short explanation of their meaning. The first eight of them were taken from [24]. The last one (*Support of different programming languages*) was added by the authors because sometimes one programming language is preferred for solving certain tasks over another, so it would be important to support as much of them as possible.

TABLE III
RELATION OF THE QUALITATIVE CRITERIA AND CODE GENERATORS

Tools qualitative criteria	*Tools*			
	IBM Rational Software Architect 8.0	*Sparx Enterprise Architect 9*	*MS Visual Studio 2010*	*Eclipse 3.7.2*
Ignore specific syntax – code generation language independence from modeling language	X	X	X	X
Powerful expression language – the easiest way of accessing model elements		X	X	X
Modular templates – the ability to keep a group of templates in one module				X
Nice syntax – better readability of the template text	X	X	X	X
Object oriented templates – polymorphism, inheritance and “this” reference support			X	X
Static type checking & IDE support – the ability to get error messages while writing the code	X	X	X	X
Extensions – the ability to attach additional modules to extend the functionality	X	X		X
Template AOP – the ability to define some architecture-specific code fragments inside the template				
Support of different programming languages	X	X	X	X

Table 3 shows that no one of the chosen tools match the template AOP criterion. However, in the authors’ opinion, it is not so important because modern operating systems are too different and sometimes it is easier to fully rewrite certain templates depending on the architecture than make them more complicated with specifying a lot of similar code fragments. Moreover, some languages, such as Java, do not depend on particular system architecture at all.

In the authors’ opinion the most important criterion is static type checking & IDE support because the earlier IDE will warn the developer about a mistake, the earlier it will be corrected. All of the tools match this criterion, which points to their good quality.

The other very important criterion is object oriented templates. This fact is supported by the core of the object oriented paradigm. However, not all of the code generators match this criterion.

Talking about extensions, Microsoft Visual Studio 2010 might also match it, however, in comparison to other tools, it poorly supports plug-ins.

Thus, the hypothesis presented at the start of this subsection was not confirmed. The tools are qualitative. They make coding and transformation writing process much easier. Moreover, they provide the user with certain useful additional functions and support a variety of programming languages.

C. Quantitative evaluation of code generators

According to the presented classification of the tools, they use a variety of different techniques to generate the source code. The quality of some of the most popular code generators was also substantiated. However, as the current subsection shows, the main problem of the code generation unpopularity in the industry lies within the code generators functionality itself.

Such popular tools as Sparx Enterprise Architect, Visual Paradigm, Microsoft Visual Studio 2010 were chosen to test their functionality. Sparx Enterprise Architect and Visual Paradigm for UML [25] are mainly recognized as UML modeling tools with the capability to generate program code, compared to Visual Studio, which is a development environment. In the authors' opinion, a tool which is a development environment could have more support for required modeling structures from the code perspective, as well as better algorithms for code generation from the models.

To test the chosen tools 69 experimental models were made. Each of them was created on the basis of the knowledge of the notation of UML elements and its semantics (from UML specifications) and, on the other hand, the knowledge about C# rules and its syntax. For code generator testing, the black-box testing strategy was chosen, which means, that the authors provided the input (UML class diagram) and analyzed the output (C# code). The authors were making experiments for the following programming structures/UML notation: access modifiers (private, protected, internal); class modifiers (abstract, sealed, static); method modifiers (static, new, override, abstract); method parameter modifiers (params, ref, out); accessor methods; multiplicity; default values; read-only and derived values; constructors; destructors; stereotypes: constant, event, property; active class; constraints: ordered, unique, redefines; naming conventions and keyword usage; namespace scope; tag-values: precondition, postcondition, etc.

The results for each of the tool (Enterprise Architect / Visual Paradigm / Visual Studio) distributes respectively: almost half with errors – compilation errors 55%/45%/39% and execution errors 3%/5%/4%, one fourth is information loss 20%/26%/16%, here we can also add missing notation category, which is 11% for Visual Studio. The missing notation category was added (to the list of categories) because of the Visual Studio, originally the experiments were done with Sparx EA and Visual Paradigm, and only later tested with Visual Studio. Because it was not possible to provide tagged values and constraints, these tests resulted in missing notation. Finally, only 14%/18%/18% of the tests appeared to be correct.

The reason for such results, which showed the analysis of three code generation tools is the primitivism of transformations. Although it is possible to change the transformation templates, this does not solve the problem completely because of limitation to local scope. The root problem lies in the simplicity of program code generators, which simply transfer the pattern of model information to the program code without any additional testing and decision making on the required information conversion for the target programming language. Generators do not have any additional knowledge support about target platform restrictions, laws and keyword combinations.

In fact, the existing tools try to implement some small pieces of different ideas – all at once. They do not focus on one specific method and do not attempt to fully complete it. The developers of code generators also do not try to bring some revolutionary new ideas in the field of this research, instead they just try, for example, to modify the template-based code generation approach. Because of this, the evolution of the code generation nearly stopped.

V. Practical experiment of Code Generator Development

The code generator evaluation test clearly showed that the code generation puzzle is not yet solved. Thus, it makes sense to experiments with the new code generator development. The authors of this paper also want to present their tool, which combines frame processing as well as templates and filtering code generation techniques. According to previously presented taxonomy, the tool is a partial-class generator and tier or layer generator. It realizes the template-based code generation approach. The presented tool was written in C# programming language and uses standard .NET libraries for parsing XML documents.

A. The ideas born while developing the tool

During the tool development process it became totally clear that it is very difficult to invent something new in the code generation field. Every idea, which at the first glance seemed original eventually reduced to template writing and iterating through model meta-data. That is why it was impossible to discard the template approach completely. However, use of templates can be quite an advantage because they provide an opportunity to develop a more universal code generator. It was also mentioned that template-based approach is more effective than the visitor-based, besides, it offers more flexibility during code specification. Moreover, in the authors' opinion, it is easier to make the code generator, which uses template-based code generation approach. However, the idea of making templates as much universal as possible, led to certain difficulties during the tool development. In the end, some share of universality was sacrificed to make certain pre-planned features available. In this aspect, it would be reasonable to mention that OMG organization itself interferes the universality by allowing modeling tools to use xmi:Extension element [26, 27]. Thus developers of modeling tools can extend the XMI document as they like and eventually they do so. Thus, every template that represents one piece of code can be overwritten several times depending on the modeling tool.

It was already mentioned earlier that with the system growth, the templates may also become more complex and less readable. The reason is that each template includes more structures, key words, conditions, etc. That is why it would be reasonable to somehow separate the definition of the template from the definition of some of the structures mentioned above. The template hierarchy must also be better structured in relation to the elements of UML model. The authors of this paper suggest frames as the mechanism of implementing the ideas mentioned above.

B. The tool major working mechanism

The developed tool has the following idea: each frame should represent one specific type of the object-oriented

elements (class, interface, method etc.). Frames store information on how to connect code templates with the model XMI meta-data. The XPath 1.0 language is used to achieve this. Each frame has its own set of attributes. Some of them are common for all frames. It means that the user must define their values on the creation of the frame. Such attributes are: name of the frame, template, prefix, postfix, separator of the similar code pieces and meta-model XPath expression. The user may also specify his own unique attributes that connect the current frame template with the system model. Another purpose of the user-defined attributes is to keep references on other frames. User-defined attribute names may be put into the template text to specify connection between the system models and the templates.

The user should define the root frame and the meta-data XML document to make it all work. The root frame should be the one, which which keeps the most abstract information about a code to generate. The tool starts the code generation process from processing the root frame. The mechanism of frame processing is the same for all frames. First of all, a value of XPath expression attribute is taken. It is used to find the matching element from the meta-data file. Then the template is processed and written to the output file. The prefix and the postfix text are also attached to the template at the beginning and at the end of the generated text. At this time the template parsing mechanism tests the template for any user-defined attribute names. The name of the matching attribute is replaced with the value from meta-data if the matching attribute keeps an XPath expression. If the matching attribute keeps reference to the other frame, its name is replaced with the generated text of the matching frame.

C. Demonstration of the tool operation

A simple example is provided further to demonstrate how the tool works. Let us imagine that we have two public classes: *Thing* and *Table*. The second class extends the first one. The class *Thing* has two private attributes of type integer: *x* and *y*. The class *Table* has the private attribute named "*color*", which is of the type *String*. The XML document, which represents this model is given below:

```
<?xml version="1.0" encoding="windows-1252"?>
<model>
  <element ID = "1" type = "class" name = "Thing"
scope = "public">
    <attribute ID = "2" name = "x" scope = "private"
type = "int"/>
    <attribute ID = "3" name = "y" scope = "private"
type = "int"/>
  </element>
  <element ID = "4" type = "class" name = "Table"
scope = "public">
    <attribute ID = "5" name = "color" scope =
"private" type = "String"/>
    <parent ID = "6" IDref = "1"/>
  </element>
</model>
```

A list of the frames, which specify code generating from this model is shown in Table 5.

TABLE IV
A LIST OF FRAMES

Attribute names	*Attribute values for each frame*		
Name	_element	_attribute	_parent

Template	_scope _type _name _parent {
	_attributes
	}
		_scope _type _name ;	_parent

XPath	//model/element	attribute	Parent
Separator	\n\n	\n	
Prefix			extends
Postfix			
_type	type	type	attribute is not defined
_name	name	name	attribute is not defined
_scope	scope	scope	attribute is not defined
_attributes	Reference on the frame „_attribute"	attribute is not defined	attribute is not defined
_parent	Reference on the frame „_parent"	attribute is not defined	FindNodeAttr IDref ID //model/element[@type='class'] name

In case of the presented example the root frame is *_element*. Thus, it is the first frame to process. Basically, the *_element* frame describes the generation of such structure as class. According to the tool's working mechanism the *XPath* attribute's expression is being processed, and in this example it points to the element called "*element*" of the XML text presented above. It means that the *_element* frame attributes *_type, _name and _scope* point to the attributes *type, name,* and *scope* of the XML text element, which name is "*element"*. The attributes *_parent* and *_attributes* point to the other frames with the names "*_parent"* and "*_attribute"*. According to the *separator* attribute, each peace of the *_element* frame template will be separated with two caret return symbols if it meets more than one element with the name "*element*" inside the XML text (actually it will, because of the example definition).

The frame *_attribute* has the structure, which is similar to the previous one. Thus, the frame *_parent* will be described in

more detail. The specifics of *XPath* attribute is that each child frame will point to the child element of the XML element its parent points to. In this example the frame *_element* points to the *_parent* frame, which means that in this case this frame will be associated with the XML element available by the path "*//model/element/parent*". The *_parent* frame has only one user-defined attribute, which name is "*_parent"*. This attribute contains usage of the function *FindNodeAttr* that finds an attribute value of specific XML element, which other attribute matches to the value of the different XML element specific attribute value. In the presented example this function is used to get the name of the class parent, if such exists. If it is true, then the *prefix* attribute's text "extends" is being added before the name of the parent.

Speaking about the functions, some of them were introduced to implement such things as searching for similar elements or testing for existence of some the particular element. To introduce these functions, the universality has to be sacrificed , which was mentioned earlier. The reason is that the first version of XPath language does not provide a number of vital functionalities, therefore in the future versions of the tool it will be replaced with the XQuery language.

The code generation process results into the following code:

```
public class Thing {
  private int x;
  private int y;
}

public class Table extends Thing {
  private String color;
}
```

VI. Conclusion

For more than ten years scientists in cooperation with big industrial organizations have been trying to find different ways of applying the Object Management Group invention – MDA architecture. If they succeeded, the coding process would become fully automated. It would lead to reducing the amount of bugs inside the source code almost to zero as well as system independency from the particular computer architecture. Unfortunately, as of today the fully functioning MDA based system remains a utopia. Statistics proves that there are some good quality MDA based systems; however, there are not many of them all over the world. That is why in the authors' opinion good work has been done by qualified and experienced human specialists rather than code generators.

The code generation field is "crowded" with numerous and differing standards, which prevents from using them in practice. However, some practical disciplines of code generation do exist. The authors of this paper found out that all of them stem from the two different code generation approaches: template-based and visitor-based. The study resulted into the taxonomy, which shows how these two approaches as well as code generation techniques and code generator types are linked together. The created taxonomy then was used to classify such tools as IBM Rational Software Architect 8.0, Sparx Enterprise Architect 9, Microsoft Visual Studio 2010 and Eclipse 3.7.2. As a result, it became clear that all those tools realize a variety of mechanisms to generate a source code, moreover they match some quality criteria presented during the research. Theoretically, it means that these tools have a really great functionality as well as they are easy to use. The latter is true. As for the code generators functionality, it has a huge hidden problem, which makes code generation so rarely used in the industry.

The functionality of such tools as Sparx Enterprise Architect, Visual Paradigm and Microsoft Visual Studio 2010 was tested on 69 different experimental models. The study showed that only 14% of the tests for Sparx Enterprise Architect, 18% for Visual Paradigm and 18% for Microsoft Visual Studio 2010 are correct. In the authors' opinion such bad results are the consequence of the primitivism of transformations and lack of some clever way of applying them. Basically, code generators do not have additional information about the target platform. According to classification, the other reason of code generators bad functionality is that they are designed to perform many different tasks by applying a lot of techniques at once.

In the authors' opinion, another weak spot are the templates. On the one hand, they are quite effective and therefore the most commonly used. On the other hand, they become too complex and difficult to read during developing thus making the system grow too big. However, the process of designing the authors' own code generating tool showed that it is difficult to invent something that would be much more innovative than the templates. The mentioned tool implements the templates in combination with the frames. Frames are used to make templates less overloaded by meta-model connection and other replaceable strings while storing it inside the attributes. Unfortunately, at present the described tool is able to transform models into the source code, however it cannot be extended to perform the opposite task. One of the most significant disadvantages of the tool is that it was not designed to update the source code, only to regenerate it every time it runs. This actually lowers the speed of performing transformations. In the authors' opinion, in this context the question of speed is not very important. As it was already mentioned, the main goal of code generators is to generate the correct code that is why it could be reasonable to sacrifice speed for functionality.

Despite a permanent solution of the code generators' functionality problem could be developing completely new approach of code generation or rewriting the existing specifications, it may consume a lot of time and resources. To operatively improve the situation with code generation it is worth searching for some effective combinations of already existing code generation methods and techniques. It could also be reasonable to tryi to develop some minor tools, which would concentrate on a narrow spectrum of tasks and use these tools in combination rather than using the bigger ones separately.

ACKNOWLEDGEMENTS

The research presented in the paper is partly supported by Grant of Latvian Council of Science No. 09.1269 "Methods and Models Based on Distributed Artificial Intelligence and Web Technologies for Development of Intelligent Applied Software and Computer System Architecture".

REFERENCES

[1] UML Unified Modeling Language Specification, OMG document , [Online]. Available: http://www.omg.org [Accessed: Sept. 23, 2012]
[2] Sejans J., Nikiforova O. Practical Experiments with Code Generation from the UML Class Diagram, Proceedings of MDA&MDSD 2011, 3rd International Workshop on Model Driven Architecture and Modeling Driven Software Development In conjunction with the 6th International Conference on Evaluation of Novel Approaches to Software Engineering, Osis J., Nikiforova O. (Eds.), Beijing, China, June 8-11, 2011, SciTePress, Portugal, Printed in China, pp. 57-67
[3] A. Bajovs, Research of the Basic Principles of the Model-To-Code Transformation, Bachelor Thesis, Riga Technical University, 2012
[4] OMG: MDA Guide Version 1.0.1, [Online]. Available: http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/MDAGuide101Jun03.pdf [Accessed: Sept. 23, 2012]
[5] M. Volter, "From Programming to Modeling - and Back Again", IEEE Software, Nov.-Dec. 2011, pp. 20-25.
[6] T. Stahl and M. Volter, Model-Driven Software Development, Wiley, 2006, pp. 428.
[7] J. L. Eveleens and C. Verhoef, "The Rice and Fall of the Chaos Report Figures," IEEE Software, IEEE Computer Society, 2010, pp. 30-36.
[8] E. Kalniņa, Model Transformation Development Using Mola Mappings And Template Mola, PhD thesis, University of Latvia, 2011.
[9] M. Brambilla, J. Cabot, and M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool Publishers, 2012, pp. 173.
[10] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process, Addison-Wesley, 2002, pp. 512.
[11] O. Nikiforova, A. Cernickins, and N. Pavlova, "Discussing the Difference between Model-driven Architecture and Model-driven Development in the Context of Supporting Tools," Proceedings of the 4th International Conference on Software Engineering Advances, IEEE Computer Society, 2009, pp. 1-6.
[12] OMG: Catalog of OMG Modeling And Metadata Specifications, [Online]. Available: http://www.omg.org/technology/documents/modeling_spec_catalog.htm [Accessed: Sept. 23, 2012]
[13] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches, OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture Anaheim OOPSLA, Canada:University of Waterloo, 2003, pp. 1-17.
[14] J. Herrington. Code Generation in Action. Manning, 2003, pp. 342.
[15] Sejans, J.: Analysis of Notational Elements of UML Class Diagram. (In Latvian: Valodas UML klašu diagrammas elementu notācijas analīze). Bachelor Thesis, defended in Riga Technical University. RTU (2007)
[16] Sparx Systems' Enterprise Architect, [Online]. Available: http://www.sparxsystems.com.au/ [Accessed: Sept. 23, 2012].
[17] IBM: Are you new to Rational? Get acquainted with IBM Rational collaborative and integrated solutions for software and systems delivery, [Online]. Available: http://www.ibm.com/developerworks/rational/newto/ [Accessed: Sept. 23, 2012].
[18] Microsoft: Visual Studio 2010, [Online]. Available: http://msdn2.microsoft.com/en-us/vstudio/default.aspx [Accessed: Sept. 23, 2012].
[19] Eclipse: About the Eclipse Foundation, [Online]. Available: http://www.eclipse.org/org/ [Accessed: Sept. 23, 2012].
[20] IBM: Rational Software Architect 8.0 Documentation, [Online]. Available: http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp [Accessed: Sept. 23, 2012].
[21] Sparx: MDA Transformations, [Online]. Available: http://www.sparxsystems.com/uml_tool_guide/mda_transformations/mdastyletransforms.htm [Accessed: Sept. 23, 2012].
[22] MSDN: Code Generation and T4 Text Templates, [Online]. Available: http://msdn.microsoft.com/ru-ru/library/bb126445.aspx [Accessed: Sept. 23, 2012].
[23] Eclipse: Documentation - Current Release, [Online]. Available: http://help.eclipse.org/indigo/index.jsp [Accessed: Sept. 23, 2012].
[24] M. Völter. Best Practices for Model-to-Text Transformations, Ingenieurbüro für softwaretechnologie, 2006, pp. 3.
[25] Visual Paradigm, [Online]. Available: http://www.visual-paradigm.com/ [Accessed: Sept. 23, 2012].
[26] OMG: MOF 2 XMI Mapping Specification Version 2.4.1, [Online]. Available: http://www.omg.org/spec/XMI/2.4.1 [Accessed: Sept. 23, 2012].
[27] A. Cernickins, O. Nikiforova, K. Ozols, J. Sejans. An Outline of Conceptual Framework for Certification of MDA Tools, Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development, In conjunction with ENASE 2010, In Janis Osis, Oksana Nikiforova, (Eds.), Athens, Greece, July 2010, SciTePress, pp. 60-69.

Andrejs Bajovs received the Bachelor degree in computer systems from Riga Technical University, Latvia, in 2012.
He is presently the first year master study's student and a scientific assistant at the Department of Applied Computer Science, Riga Technical University.
His current research interests include original ways of code generation in model driven architecture.
E-mail: andrej_bv@inbox.lv

Oksana Nikiforova received the doctoral degree in information technologies (system analysis, modeling and design) from Riga Technical University, Latvia, in 2001.
She is presently a Full Professor at the Department of Applied Computer Science, Riga Technical University, where she has been on the faculty since 1999. Her current research interests include the object-oriented system analysis and modeling, especially the issues in the framework of Model Driven Software Development (MDSD). She has published extensively in these areas and has been awarded several grants. She has participated and managed several research projects related to the system modeling, analysis and design, as well as participated in several industrial software development projects.
She is a member of RTU Academic Assembly, Council of the Faculty of Computer Science and Information Technology, RTU Publishing Board, RTU Scientific Journal Editorial Board, etc. She is a co-chair of workshops focused on MDSD – MDA 2009 in conjunction with ADBIS, MDA&MTDD 2010 and MDA&MDSD 2011 in conjunction with ENASE. She was awarded as RTU Young Scientist of the Year 2009.
E-mail: oksana.nikiforova@rtu.lv

Jānis Sējāns has engineering science master degree (Mg.sc.ing) in information system field from Riga Technical University, in 2010. Currently studying at the Faculty of Computer Science and Information Technology as Phd student.
He is a researcher in the Department of Applied Computer Science of Riga Technical University. The work experience is related to ERP system programming and implementation, since 2002. Currently running own company TownTech and working at Joint Stock Company Latvian Roadworks as a programmer for ERP system.
His current research interests include model driven software development, especially its issues in code generation improvement.
E-mail: janis.sejans@rtu.lv