Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2011
Volume 47

Problems and Perspectives of Code Generation from UML Class Diagram

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

***Abstract* – As a result of increasing technological diversity, more attention is being focused on model driven architecture (MDA), and its standard – Unified Modeling Language (UML). UML class diagrams require correct diagram notation mapping to target programming language syntax under the framework of MDA. Currently there are plenty of CASE tools which claim that they are able to generate the source code from UML models. Therefore by combining the knowledge of a programming language, syntax rules and UML class diagram notation semantic, an experimental model for stressing the code generator can be produced, thus allowing comparison of quality of the transformation result. This paper describes a creation of such experimental models.**

Keywords: code generation, MDA transformation, model-to-code transformation, UML class diagram.

I. Introduction

The current problem the IT industry faces is the growing diversity of technologies, which are becoming increasingly complex, while at the same time not providing solutions for product quality and development process improvement. The reason is the lack of adequate tools which would be able to hide technological complexity and permit the management of products from a higher level of abstraction. Thus, the software development process is focused on the coding aspect – "how?", instead of focusing on the problem domain – "what?". It should be mentioned that "traditional" CASE tools do not address this problem. The effort involved in drawing up models is nothing more than a supplement to the documentation, because the next activity involves a manual transformation of the model element notation into a software product (program code). This "semantic gap" between the model notation and the programming language raises additional challenges for systems engineers, who need to maintain not only the software product, but also a model. Because of requests for changes it is practically impossible to avoid direct changes in the program code that bypass the system model. As a result, this leads to an inadequate model [1].

Therefore, more attention is being focused on Model-Driven Development (MDD), which includes the idea of software development with systems modeling from a higher level of abstraction, using Model Driven Architecture (MDA). MDA is based on model transformations, starting with a computational independent model (CIM) and ending with the platform specific model (PSM). The subsequent transformation leads to a programming language code.

Models are created using the Unified Modeling Language (UML) [2]. UML is a set of diagram notations, which helps to describe the system design – its structure, behavior and interaction. One of the UML diagrams is a class diagram, which is the basis for system structure modeling and, according to the principles of MDA, makes program code generation possible. [3] describes UML class diagram elements and their notation in the context of language evolution. Additionally the author of the study has shown that there are modeling tools that fully support notation defined in UML 2.0 [4], for example Visual-Paradigm for UML. Of course there are several different approaches to system structure modeling, the authors, namely, Grady Booch, James Rumbaugh and Ivar Jacobson, are also UML notation founders. Their analysis and most of object-oriented system analyses and structural modeling approaches are described in the previous paper on this issue [5]. However, [3] survey of Latvian IT companies with an aim to identify usable notation in class diagram, as well the survey [6] shows that the class diagram is still not widely used, for program code generation or system documentation. Thus, class diagram losses its role in software development – to solve the "semantic gap" and to serve as a "bridge" between the problem space and the solution space [1].

The goal of the paper is to develop experimental models of the UML class diagram that contain semantically correct and erratic structures, with the aim of determining the quality of the code generator in UML modeling tools and generation support tools. The goal can be achieved by combining knowledge of UML class diagram element notation on the one hand and knowledge of the expected result on the other hand. In this paper the authors focus on insights into the experimental model as well as demonstrate its usage in one of the modeling tools and examine the quality of the obtained program code (quality of code generator).

Currently there are several modeling tools that include code generation options. In addition a variety of programming languages such as Java, C#, C++, Ruby, Delphi, Perl, Python, Objective-C, Ada and PHP can be chosen as a transformation target platform. In this paper, the authors examine the transformation of class diagrams into the programming language C#.

According to [7] study, which compares five modeling tools: ArgoUML, Altova UModel, Sparx Enterprise Architect, Rational Rose Enterprise and MyEclipse Enterprise Workbench, the best support in the context of the idea of MDA is considered to be Sparx Enterprise Architect. Given that the Sparx Enterprise Architect v7.5.845 (further SPARX) supports the code generation for C#, it has been selected for model transformation experiments. As a basis for the C#

programming language syntax and structures the .NET 3.5 is considered, as well as new structures from .NET 4.0.

The paper is structured as follows: The following section gives an overview of basic transformations of class element into programming code and analysis of transformation results in comparison with the code expected within the rules and statements of object-oriented programming. The third section shows several experiments with transformation of different types of classes into programming constructions and presents an analysis of the quality of the transformation result achived in these experiments. Modeling construction of class attributes and methods are presented in the fourth and fifth sections respectively. These models provide possibility to check the correctness of the transformation results according those class elements, which relate to class structure and behavior. In conclusion, several findings of the present research are discussed.

II. Transformation Experiments with Basics of the UML Class Element

UML notation for class element contains information about the problem domain unit (name), its characteristics (attributes) and operations (methods). An operation is a procedure declaration, whereas a method is the body of a procedure (implementation). In this paper these terms will be used interchangeably, due to the mixing model and code point of view (where method is preferred), except when it is important to be precise about the difference. Class element notation options are shown in Fig. 1.

Fig. 1. Alternative notations of the UML class element

The formal syntax for class declaration in programming language C# [8]:

```
[attributes]
[class-modifiers] class identifier [:class-base-clause]
{class-body}[;]
```

From the formal syntax of class declaration it follows that other parameters such as class metadata and access modifiers can be specified in addition to the class name. Programming language C# defines five accessibility levels (access modifiers): **public**, **private**, **protected**, **internal** and **protected internal**, these can be used with classes, attributes and methods. According to [9], the class has the following laws-restrictions for access modifiers:

1. Classes that are declared directly within a namespace can be either **public** or **internal**. Internal is the default if no access modifier is specified. Even if the namespace is not provided, the class still belongs to **System** namespace, thus the access modifier should be **public** or **internal**. Class members including nested classes can be declared with any of the five access modifiers.

2. Derived classes (subclasses) cannot have greater accessibility than their base types. For example, class B cannot be **public** if it is derived from **internal** class A. If this were allowed, it would mean that class A should be **public**, because all **protected** or **internal** members are accessible from the derived class B.

To verify class access modifier transformation both of these restrictions are modeled (ProtectedClass and inheritance SuperClass←SubClass), as well as one **internal** class, which includes four nested classes with different access modifiers (Table 1 right column). This experiment allows the researchers to find out if there is any treatment for access modifiers in the modeling tool or it performs plain keyword transfer.

III. Transformation Experiments with Different Types of Classes

The class is a container type that can be instantiated, so it describes common properties and behaviors of its instances – objects. Of course there are also other structure types such as **struct**, **enumeration**, **array**, **interface** as well as different class types like **abstract**, **sealed**, **static** etc. Each of these types has their own features and is used for different purposes, thus promoting the rules for syntax, context and appropriate usage - semantics. It is possible to create an erratic model for every type, however, in this paper, the authors describe semantic and experiment results for three class types: **sealed**, **active** and **abstract**. The **sealed** and **abstract** classes are used in inheritance hierarchy and thus they have limitations and conditions for identification, which can be examined. The term **active** class comes from UML specification, where the semantics for this type of class is provided, and thus can also be a subject for testing the transformation. In the following subsections, the conditions that must be met by modeling tools while generating the code are described. The expected result and transformation result are also provided.

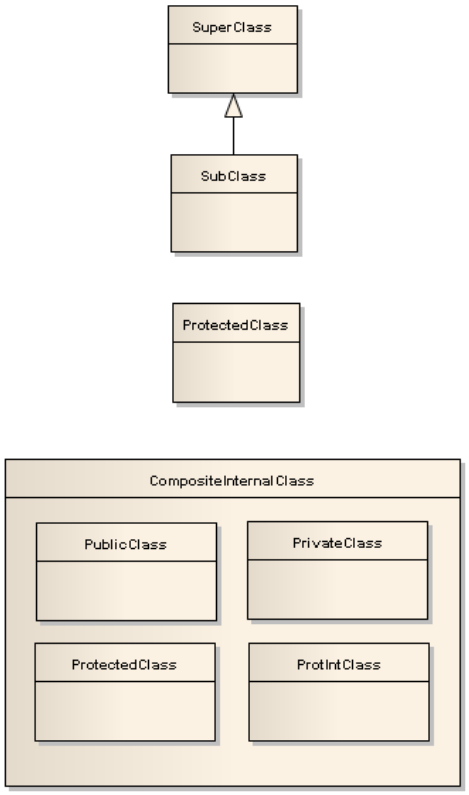
A. Sealed class transformation

Inheritance hierarchy can be limited using a sealed class, which is a leaf class that prevents further inheritance. UML defines the following class with the keyword **leaf**, while in the programming language C# the keyword **sealed** is used. Both keywords describe identical semantics, thus **leaf** class should be transformed into **sealed** class. In other words, the generated program code should contain keyword **sealed**.

Although [4] does not define notation for leaf class, the SPARX tool shows tag value **{leaf}** in the class name compartment (Table 2 right column). Contrary to leaf class semantics, the experimental model will contain SubClass that is inherited from SealedClass. Whereas this construction is prohibited, in the transformation process the **leaf** tag should be ignored and the keyword **sealed** should not appear in the generated program code. Another scenario, the modeling tool should not allow a class to be derived from leaf class.

TABLE I

TRANSFORMATION RESULT FOR ACCESS MODIFIERS

Model	Expected code	Transformation result
	`namespace Access_modifiers { internal class SuperClass { } internal class SubClass : SuperClass { } }`	`namespace Access_modifiers { internal class SuperClass { public SuperClass(){ } } public class SubClass : SuperClass { } }`
	`public class ProtectedClass { }`	`protected class ProtectedClass { }`
	`internal class CompositeInternalClass { public class PublicClass { public PublicClass(){ } } private class PrivateClass { public PrivateClass(){ } } protected class ProtectedClass { public ProtectedClass(){ } } protected internal class ProtIntClass { public ProtIntClass(){ } } public CompositeInternalClass(){ } }`	`internal class CompositeInternalClass { public class PublicClass { public PublicClass(){ } } private class PrivateClass { public PrivateClass(){ } } protected class ProtectedClass { public ProtectedClass(){ } } protected internal class ProtIntClass { public ProtIntClass(){ } } public CompositeInternalClass(){ } }`

The transformation results are shown in Table 2. The obtained results show that semantics were not considered, either in the model or in the transformation result, despite the fact that it is defined in UML, as well as in the programming language. It should be mentioned that notwithstanding the restriction in SPARX tool while inheriting from sealed class, the limitation is not complete since it is possible to create inheritance and afterwards point out that the base class is a sealed class.

As a result, the warning does not appear and an incorrect program code is generated. With this experiment it is shown that the tool is able to do model validation and disallow incorrect constructions, however, the validation should not depend on the sequence of actions, thus allowing the validator to be by-passed. Finally, even if the modeling tool allows the model to be created with a wrong keyword then it should be ignored in the transformation process and should not appear in the code. In contrast, the modeling tool generated the program code that does not compile.

B. Active class transformation

An active class owns an execution thread so that all instances of that class, or the active objects, can initiate control activity. An active object can execute an action in parallel with other active and passive objects, and initiates actions by itself, in its own code or by sending messages.

Terms active class and active object are used in real-time systems, where the action takes place in a multi-thread environment. In fact for active objects it is not possible to predict when the specific behavior will be triggered (asynchronous) while for passive objects this may be determined by the message flow (synchronous) [10].

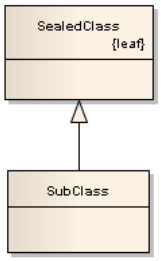
Similarly, OMG "UML Superstructure Specification v2.2" [4] mentions that active class semantics is related to object execution in a separate thread, hence the code generator should create a class construction that is designed to be executed in a separate thread.

The expected code for active class transformation into programming language C# is to use ***Threading*** (`using System.Threading`), because C# does not require any special interface or inheritance from the base class, and that enables an instantiation of object in a separate thread. In contrast, the programming language Java should derive from Thread (**extends Thread**) and implement **Run** method.

TABLE 2

TRANSFORMATION RESULT FOR SEALED CLASS

Model	Expected code	Transformation result

	`public class SealedClass { }`
	
	`public class SubClass : SealedClass { }`
		`public sealed class SealedClass { }`
		
		`public class SubClass : SealedClass { }`

However, in a similar manner, it is expected that the generated class will contain the **Thread** attribute, class constructor which initiates this attribute and **Run** method, which starts the thread. The expected code and transformation result is shown in Table 3.

The transformation result of active class allows us to conclude that the SPARX tool has no semantic mapping for the isActive parameter, because the generated code for an active class is the same as for a simple class. On the one hand, it can be argued that the modeled class itself does not contain **Thread** attribute and **Run** method. On the other hand, if the attribute and method were provided, then it is not clear, what the difference between simple class and active class is, so it results in loss of notation meaning, because the outcome was not different. For the sake of judgment objectivity, it was examined whether the active class transformation into the Java programming language shows the difference (should inherit from **Thread**). However, the Java code did not contain "**extends Thread**". In conclusion, the code generator must have knowledge of special attributes or methods that are needed for specific construction to ensure specified semantics.

TABLE 3

TRANSFORMATION RESULT FOR ACTIVE CLASS

Model	Expected code	Transformation result

	`using System;`
	`using System.Threading;`
	`namespace CodeGenTest`
	`{`
	`  public class ActiveClass`
	`  {`
	`  Thread aThread;`
	`  public ActiveClass()`
	`  {`
	`  if (aThread == null)`
	`  aThread = new Thread(new ThreadStart(ThreadJob));`
	`  }`
	`  public void Run()`
	`  {`
	`  if (aThread.IsAlive == false)`
	`  aThread.Start();`
	`  if (aThread.ThreadState == ThreadState.Suspended)`
	`  aThread.Resume();`
	`  }`
	`  public void Pause()`
	`  {`
	`  aThread.Suspend();`
	`  }`
	`  private void Cleanup()`
	`  {`
	`  aThread.Join(0);`
	`  aThread = null;`
	`  }`
	`  public void Quit()`
	`  {`
	`  Cleanup();`
	`  }`
	`  protected virtual void ThreadJob()`
	`  {`
	`  }`
	`  }`
	`}`
		`namespace CodeGenTest {`
		`  public class ActiveClass {`
		`  public ActiveClass()`
		`  {`
		`  }`
		`  }`
		`}`

So, in cases where these attributes or methods are not identified in the model, they should be generated. A concrete example with an active class is one of the cases that indicates a loss of information and not a lack in notation, because the active class indicator is not transferred to the code level.

C. Abstract class transformation

Abstract class is the class that cannot be instantiated. Typically, an abstract class has one or more abstract operations. Thus a class that has at least one abstract operation must by definition be an abstract class. An abstract operation has no implementation method, only the declaration.

Abstract classes are used in inheritance hierarchies, where the abstract class is superclass, while the subclass inherits abstract operations, and in the case where it is not another abstract class, it must implement all abstract methods (**override** is required). According to [9] and [11], abstract classes have the following laws – conditions:

1. If a class has at least one abstract method, it must be an abstract class, thus the definition must contain the keyword **abstract**.
2. A class that is derived from an abstract class must redefine all abstract methods using the keyword **override**.
3. Because an abstract method declaration provides no actual implementation, the method declaration must not contain braces ({ }).
4. Because the abstract method must be redefined in a derived class, the access modifier cannot be **private**.
5. While redefining the method, the access modifier must not be modified, for example, if an access modifier in a superclass is **public**, then it may not be **protected** in a subclass, since it has to be **public**.
6. It is an error to use the keywords **static**, **virtual**, **sealed** and **new**, together with the **abstract** keyword.

For abstract class transformation analysis, a model which violates all the mentioned laws is created and it contains the following:

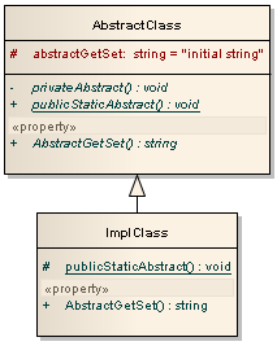
1. The class will contain an **abstract** method, but at the same time it will not be marked as abstract, so the code generator must understand that if the class has at least one abstract method, then the whole class is abstract. In addition, the access modifier for the abstract method will be **private**, so in this case the code generator must understand that this is a prohibited construction. The method name is privateAbstract.
2. The second abstract method (publicStaticAbstract) will be marked as static. Because this is a prohibited construction, the keyword **static** must be ignored.
3. The derived class will not have privateAbstract method, however, the law remains and code generator should redefine the method. Whereas the publicStaticAbstract will be redefined in the model, but with a different access modifier - **private**. The code generator should ignore this access modifier and reassign it, so it is the same as a superclass access modifier – **public**.
4. In order to verify the correct use of braces, the attribute abstractGetSet is created. It is supposed that the code generator will create abstract get and set methods, as well as correctly redefine them in a derived class.

The model, expected result and transformation result are shown in Table 4. Because the model contains conflicting parameters, it will show inconsistent notation, for example, private abstract class will be *–privateAbstract* (the minus sign as a prefix and method name in italics), which is incorrect from the code perspective.

TABLE 4

TRANSFORMATION RESULT FOR ABSTRACT CLASS

Model	Expected code	Transformation result

	`public abstract class AbstractClass {`
	`  protected string abstractGetSet = "initial string";`
	`  public abstract string AbstractGetSet {`
	`  get;`
	`  set;`
	`  }`
	`public abstract void PrivateAbstract();`
	`  public abstract void PublicStaticAbstract();`
	`}`
	`public class ImplClass : AbstractClass  {`
	`  public override string AbstractGetSet {`
	`  get {`
	`  return abstractGetSet;`
	`  }`
	`  set {`
	`  abstractGetSet = value;`
	`  }`
	`  }`
	`  public override void PublicStaticAbstract()`
	`  {`
	`  }`
	`  public override void PrivateAbstract()`
	`  {`
	`  }`
	`}`
		`//missing abstract - compilation error`
		`public class AbstractClass {`
		`  protected string abstractGetSet = "initial string";`
		`public abstract string AbstractGetSet{`
		`  get;`
		`  set;`
		`  }`
		`  private abstract void privateAbstract();`
		`  public abstract static void publicStaticAbstract();`
		`}`
		`public class ImplClass : AbstractClass {`
		`  public override string AbstractGetSet{`
		`  get{`
		`  return abstractGetSet;`
		`  }`
		`  set{`
		`  abstractGetSet = value;`
		`  }`
		`  }`
		`  protected static override void publicStaticAbstract(){`
		`  }`
		`  //missing PrivateAbstract redefinition`
		`}`

However, that is the purpose of the experiment, to make sure that despite the incorrect model (if a tool allows one to be created), the program code is still as correct as possible, and of course, because the model should be parsed (in the transformation process) according to the target language laws.

The generated program code shows very poor results. Most of the tests failed. As previously mentioned, there are two options, either a modeling tool having some kind of validator-verificator that controls the model during the creation, thus allowing the creation of only consistent constructions, and if the designer tries to use a prohibited keyword or construction, the tool forbids such action, thus providing a correct model and correct notation.

Another option is to allow in the model any keyword combinations and constructions, but then the validator-verificator is activitated in transformation process, with the aim of providing maximal correctness according to the target transformation platform by modifying the result. In any case, the transformation must not be a primitive transfer of information, as was demonstrated by the SPARX tool. Without any additional parsing, testing and modifications the transformation result and code generator loses its value.

The reason for errors is the sketchy processing of the input model and modification irregularities, however, they must exist. The code generator must generate class structures and in the case of abstract classes, the derived class must have redefinition of all abstract methods. Even if these methods are not directly presented in the model, they must be redefined anyway, because that is the rule, otherwise the construction is incorrect. For correct model processing, the validator-verificator must be active while creating the model, as well as during the transformation process.

In conclusion to this chapter, the transformation result contains many statements which lead to compilation errors, this is caused by the SPARX tool`s incorrect method of code generation – transferring all the information exactly as specified in the model. This means that the code generator must be improved taking into account the knowledge of target platform restrictions, terms and conditions of creating constructions, because the real error rate is much higher. The authors intentionally do not deliberately describe false strings of keywords that would lead to compilation error. While analyzing the transformation results, the authors concluded that the tool has no intelligence in the code generator engine and the model is perceived to be correct, even without modeling any complex structures that actually should be processed correctly.

IV. Transformation Experiments with Class Attributes

Class attributes are displayed in a separate compartment within a class, called an attribute compartment. UML notation does not require the display of all attributes of class or even the attribute compartment itself (the experimental model for class attributes is shown in Fig. 2).

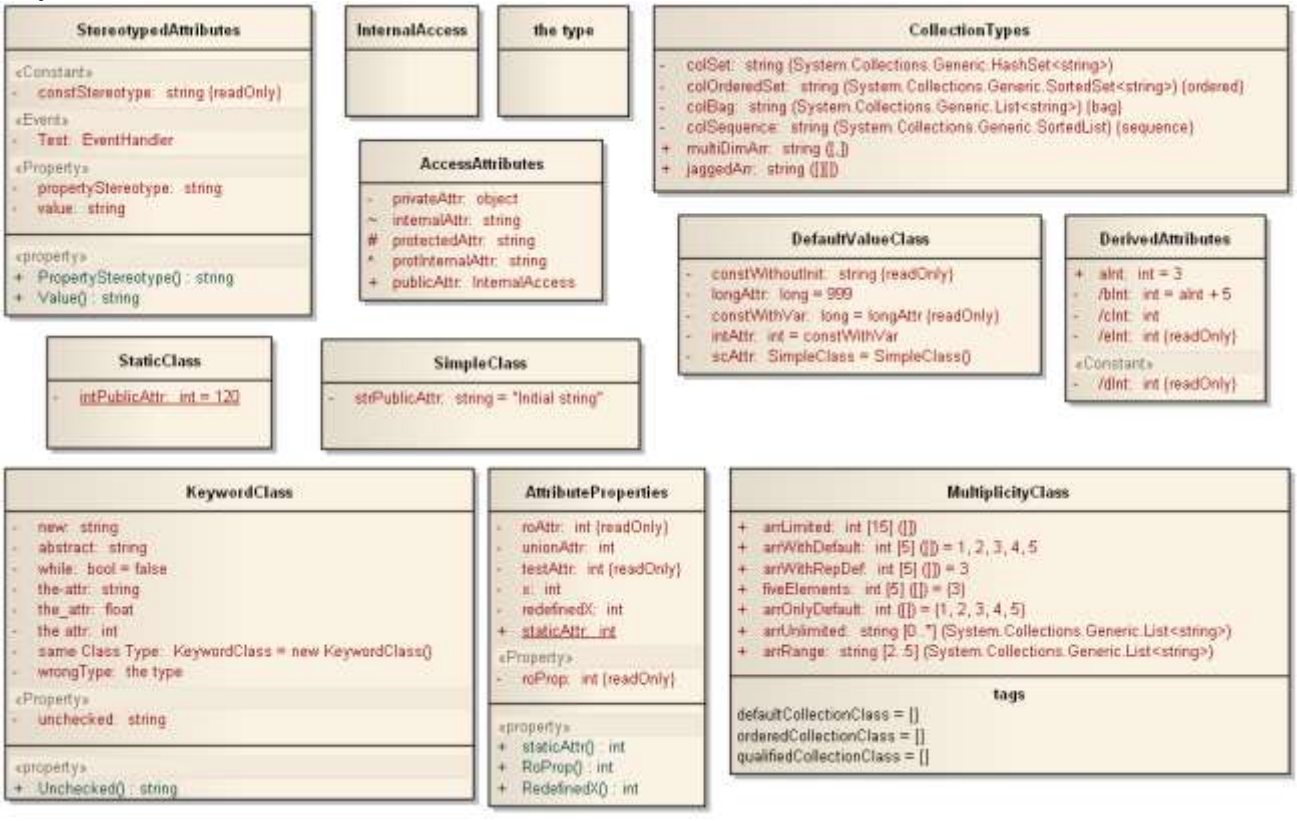


Fig. 2. Experimental model for transformation analysis of class attributes

However, UML describes detailed formal syntax for the description of an attribute:

```
[<<stereotype>>] [visibility] [/] name [: type-name]
[multiplicity ordering] [= default-value]
[{property-string}]
```

Due to the paper length limit, the authors do not explain all the restrictions and constraints that are defined for the class attributes, and which must comply with the code generation tool. More information on this is described in [12]. Instead the summary of the study results that were obtained by analyzing the semantics of class attributes and transformation result will be presented.

The model contains 32 tests for stressing the code generation tool. The examination of the SPARX tool showed that over a third of the tests caused a compilation error, one-third of the structures were defective due to the loss of the information and only 7 tests corresponded to expected result code.

Performed tests demonstrate that the code generators have no additional logic or algorithms for processing the provided keywords according to programming language syntactic rules. More generally, any combination of conflicting keywords are transferred to the code, thus creating a compilation error. This indicates that the code generator does not take into account the context and combinations, but relies on the specified model. The SPARX tool does not even make any modification regarding attribute names, so the reserved keywords as well as invalid characters are transferred into the result code, leading to error. The same happens with access modifiers and default values. They are not modified according to programming language concepts and thus all the conflicts are reflected in the result code. Another problem is derived values, multiplicity ranges, properties: **unique**, **ordered**, **union**, **redefines<x>**, which are not represented in the code nor in comment form, nor in functional. Therefore, the UML extensibility mechanism – tagged values, belong to the "loss of information" category, although the specification describes semantics for these keywords.

V. Transformation Experiments with Class Methods

Behavior and functionality of class are described in methods, which are listed in the method compartment under the attribute compartment. UML notation does not require the display of all methods of class (for example, get and set methods) or even the method compartment itself. The method signature consists not only of the method name, it also contains a list of parameters required for calling a method. The parameters are included in parentheses following the method name. For each parameter the type and the direction (**in**, **out**, **inout**) can be specified. The type is specified for the method itself in case it returns value. The formal syntax for a method in UML [10] is:

```
[<<stereotype>>] [visibility] name ([parameter-
list]) [: return-result] [{properties}]
[parameter-list] -> [direction] parameter_name [:
type-expression] [multiplicity] [= default-value]
[{property-string}]
```

The formal syntax for method declaration in programming language C# [13] is:

```
method-modifiers returntype methodname(formal-list)
{method-body}
formal-list -> parameter-modifier type parametername
               params type[] parametername
```

The experimental model for class methods contains 25 tests and it is shown in Fig. 3. Due to the limited scope of the paper, the authors do not explain all the restrictions and constraints that are defined for the class methods, and which must comply with code generation tool. More information on this is described in [12]. Instead the summary of the study results will be described. The examination of the SPARX tool showed that more than half of tests caused a compilation error, one-third resulted into information loss and only 4 tests succeed – corresponds to the expected result.

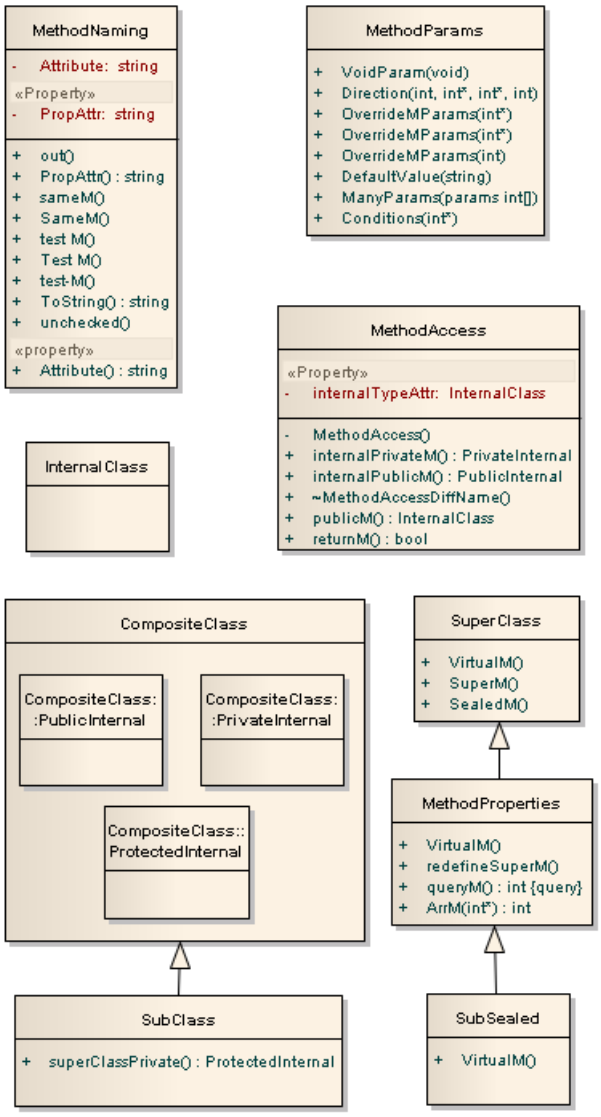


Fig. 3. Experimental model for transformation analysis of class methods

Similarly to the situation with class attributes, the transformation of class methods showed that the SPARX tool does not perform any additional modifications and thus all names, keywords and invalid characters are transferred to the result code. Consequently, similar problems are observed with methods, like failure to comply with access modifier restrictions as well as property semantics are not reflected in program code. It should be noted that during the experiment some limitations of UML notation support in modeling tools were also revealed. For example, the SPARX tool does not allow the assignment of properties for methods, despite the UML specification. As well, the transformation results show that the tool does not support point notation to access nested class structures. Instead, the program code refers to class name, which results in a compilation error.

The SPARX tool shows good results regarding the method parameter direction by using keywords **ref** and **out**. Also the SPARX tool transfers some part of model information in comment form (at least), thus retaining **bodyCondition** (for method). While this is not always true, because the **preCondition** and **postCondition** are lost during the transformation. Finally, the tool has problems in identifing collection type, either **HashSet<T>** or **SortedSet<T>**. Instead of allowing the choice of wheather to use arrays **[]**, **ArrayList** or **List<T>**, the SPARX tool requires the designer to provide each time a keyword which must be used, and as a result, the generated code contains double type. For example, if **List<string>** is provided, the result contains both types the **string** and **List<string>**.

VI. Conclusion

Since the beginning of eighties, a lot of model and code generating software systems have been offered to solve the problems regarding software productivity and quality. CASE tools developed in that time were oversold on their "complete code-generation capabilities" [14]. Nowadays, similar arguments are exposed to MDA [15], using and integrating UML models [2] at different levels of abstraction. Manipulation with models enables software development automation within CASE tools supported by MDA [16], [17].

Nowadays, when software becomes more sophisticated and complex, the industry becomes more and more challenging [18], whilst the development objectives remain the same, i.e. maintain the quality level, provide reliability, meet customer expectations, respect deadlines. According to the Standish Group study report [18], only 29% of projects have succeeded in 2004. Software development companies are losing time in routine, doing the same steps, same tests and decisions over and over again. Model-based approaches can give the next level of abstraction whilst models can be explicitly manipulated through meta-modeling. This is the way that Model Driven Development (MDD) can be applied to software development [19], [20].

For the support of MDD, there exist a lot of so-called UML diagramming tools, which give the possibility of going through all the steps of identification of elements for a class diagram [21]. As for selection the tool for code generation from class diagram, we need a possibility to get a class diagram with appropriate generated software components in the form that is suitable for further software development. The authors conduct experiments with the SPARX tool, positioned as MDA/MDD support tool [21], and have detected several inadequacies between the expected code and code generated by the tool. Unfortunately current experiments with the modeling tool that generate program codes from UML class diagram show weak and unsatisfactory results compared with the expected.

The results of the analysis described in this paper indicate that the level of the code generated by the SPARX tool is very poor and unsatisfactory. The experiment included 69 tests related to the class element, its attributes and methods. The result: ~61% of the cases contained compilation errors, ~4% had execution errors, ~19% showed loss of information. Thus, in only 15% of the tests, the generated program code may be considered to be correct (the total percentage exceeds one hundred, because the same test includes more than one category of result). Considering that the created experimental models do not encompass all possible constructions for UML class diagram, for example, various association and relationship elements, the results obtained can still be viewed as one of the main reasons why the UML class diagram is not widely applied in the IT industry.

The authors have identified a number of problems, which can be generally divided into two groups:

- modeling tools allowing the creation of improper element constructions and the use of incompatible keyword combinations that lead model transformation to incorrect codes that cannot be compiled
- generated code does not correspond to notations and details used in the model, which leads to loss of information in the result code

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. Some tools like SPARX Enterprise Architect have a code template editor with built-in transformation templates, which can be modified to support custom needs, but this does not solve the problem of lack of base information about the target platform, because restrictions may be needed for combination of elements and not one-to-one element mapping. The second mentioned group points to the complexity of the generators negligence, because the result program code does not represent appropriate constructions for semantics used in the model, resulting in loss of information and the devaluation of the work which was invested to provide additional details in the model.

This research is of both practical and theoretical importance. The practical significance is in the tool inspection and designating errors as well as indicating recommendations for improving the tool. The theoretical importance is due to the fact that the developed experimental models can be

transformed into interchangeable XMI format, thus allowing comparision and certification of modeling tools in the code generation task. The generated result can be compared to the expected one, from programming language logic, syntax and semantic points of view. Finally, research results raise a discussion on possible reasons for gaps in code generators and potential solutions to problems.

The study has shown that the tools generate different results and information must be repeated in the target programming language terms, despite the fact that it should flow from the UML notation semantics. If the MDA goal is based on the UML language, then it must certainly be based on its precise terms, structures and clearly defined semantics, which of course must be supported in the modeling tools. The current situation and sources indicate that every designer can think and use the desired keyword combinations at his/her own discretion, even if they conflict with each other. Even worse, the functionality to support UML extension mechanisms - stereotypes and tagged values, loses its value, because this information is lost during the transformation, even if the artifacts are defined in UML specification (not custom stereotypes or tagged values). Based on the semantics of UML notation, the tools with code generation capabilities must have built-in knowledge for supported target platforms, their syntax, restrictions and laws. In addition, the research has shown that tools must support not only the transformation itself, but also the possibility to validate the model, because in several cases, the adjustments can be enforced through a model. Thus, the tools must have a built-in validator-verificator which is active during the model creation as well as in the transformation process.

The authors predict that in the evolution of the transformation framework for model driven tools, it will be necessary to establish MDA support libraries for each target programming language, in order to provide additional structures and functional standards of abstractions. It will also be necessary to provide templates that meet the current evolution of development frameworks and improvements in the programming language syntax. Thus, supporting component based development can result in a tool which could generate truly functional software components. It should be noted that the programming language and technologies evolve in parallel with modeling tools, therefore it is not sufficient to choose just the target programming language. One must also select the version of the target platform, because the model transformation result will be different for different versions of the same programming language context.

The described tests and transformation analyses outline the current challenges in code generation from models, so the tests can serve as a recommendation set for code generator algorithm upgrades. The authors` vision for future research is the improvement of code generators by implementing UML semantics, restrictions and syntax rules of programming language C#, creating a plug-in, in one of the software development environments, such as Visual Studio 2010. Prior to that, code generation results in several other MDA supported tools, as well as in Visual Studio 2010 Visualization and Modeling Feature Pack will be explored.

REFERENCES

[1] O. Pastor and J.C. Molina, *Model-Driven Architecture in Practice: A Software Production Enviroment Based on Conceptual Modeling*, Springer, 2007, pp. 302.
[2] "OMG Unified Modeling Language", [Online]. Available: http://www.uml.org [Accessed : Sept. 20, 2010].
[3] J. Sejans, "Analysis of Notational Elements of UML Class Diagram," (In Latvian: Valodas UML klašu diagrammas elementu notācijas analīze) Bachelor thesis, *Riga Technical University*, 2007.
[4] „UML Superstructure Specification v2.2", February 2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/ [Accessed: Aug. 23, 2009].
[5] O. Nikiforova, J. Sejans and A. Cernickins, "The Role of UML Class Diagram in Object-Oriented Software Development," *RTU Scientific Journal, Applied Computer Systems*, 2011.
[6] B. Dobing, J. Parsons, *Dimensions of UML Diagram Use: A Survey of Practitioners*, Springer-Verlag, 2006, pp. 35.
[7] K. Ozols,"Analysis of UML code generation tools in the task of software development," (In Latvian: UML koda ģenerēšanas rīku lietošanas analīze programmu sagatavju izstrādei) Master thesis, *Riga Technical University*, 2009, pp. 142 .
[8] J. Liberty and B. MacDonald, *Learning C# 2005*, 2nd Edition, O'Reilly Media, 2006, pp. 560.
[9] „The Microsoft Developer Network", [Online]. Available: http://msdn.microsoft.com [Accessed: Oct. 7, 2010].
[10] H.E. Eriksson, M. Penker, B. Lyons and D. Fado, *UML 2 Toolkit*, Wiley, 2003, pp. 511.
[11] T. Nash, *Accelerated C# 2005*, Apress, 2006, pp. 432 .
[12] J. Sejans, "Analysis of Transformation Result from UML Class Diagram," (In Latvian: UML klašu diagrammas transformācijas rezultāta analīze) Master thesis, *Riga Technical University*, 2010.
[13] P. Sestoft and H. I. Hansen, *C# Precisely*, The MIT Press, 2004, pp. 214.
[14] J. Krogstie, "Integrating enterprise and IS development using a model driven approach," presented at 13th International Conference on Information Systems Development—Advances in Theory, Practice and Education. New York: Springer Science+Business media, 2005, pp. 43—53.
[15] A. Kleppe, J. Warmer and W. Bast, *MDA Explained : The Model Driven Architecture – Practise and Promise*, Addison Wesley, 2003.
[16] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Boston, MA: Addison-Wesley, 2002.
[17] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, 2003.
[18] R. Bendraou, P. Desfray, M. Gervais and A. Muller, "MDA Tool Components: a proposal for packaging know-how in model driven development," *Software and Systems Modeling*, vol. 7, no. 3, Berlin: Springer, 2008, pp. 329—343.
[19] A. Derezińska and R. Pilitowski, *Realization of UML Class and State Machine Models in the C# Code Generation and Execution Framework*, Institute of Computer Science, Warsaw University of Technology, 2009.
[20] D. Gasevic, D. Djuric and V. Devedzic, *Model Driven Engineering and Ontology Development*, 2nd edition, Springer, 2009.
[21] O. Nikiforova, A. Cernickins and N. Pavlova, "Discussing the Difference between Model Driven Architecture and Model Driven Development in the Context of Supporting Tools: Projection of Two-Hemisphere Model into Component Model of MDA/MDD" presented at 4th International Conference on Software Engineering Advances. IEEE Computer Society, Conference Proceedings Services, 2009, pp. 1—6.

Janis Sejans obtained engineering science master degree (Mg.sc.ing) in information systems from Riga Technical University in 2010. Currently he is a PhD student at the Faculty of Computer Science and Information Technology.

He is a researcher at the Department of Applied Computer Science of Riga Technical University. The work experience has been related to ERP system programming and implementation since 2002. Currently he is running his own company TownTech and working at Joint Stock Company Latvian Roadworks as a programmer for ERP system.

He was awarded and included in Riga Technical University Gold Student Fund in 2010.

Oksana Nikiforova received engineering science doctor's degree (Dr.sc.ing) in information technologies sector (system analysis, modeling and designing, sub-sector) from the Riga Technical University, Latvia, in 2001.

She is presently a full professor at the Department of Applied Computer Science of Riga Technical University, where she has worked since 1999. Her current research interests include object-oriented system analysis and modeling, especially related issues in the framework of Model Driven Architecture.

In these areas she has published extensively 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 awarded as RTU Young Scientist of the Year 2009.

Jānis Sējāns, Oksana Ņikiforova. Problēmas un perspektīvas koda ģenerēšanā no UML klašu diagrammas
Rakstā ir apskatītas vienotas modelēšanas valodas (angl. Unified Modeling Language – UML) klašu diagrammas elementu struktūras un to transformācija programmēšanas valodas C# kodā. Par pamatu diagrammas elementu notācijai tiek ņemts UML 2.0 specifikācijas standarts, kā arī šī gada februārī objektu vadības grupas (angl. Object Management Group – OMG) publicētie UML 2.2 superstruktūras un infrastruktūras dokumenti. Balstoties uz elementu semantiku un C# programmēšanas valodas sintaktiskajiem likumiem, tiek spriests par vēlamo transformācijas rezultātu un nosacījumiem, kas ir jāievēro koda ģenerētājam. Transformācijas rezultāta analīzei ir izstrādāti eksperimenta modeļi, kuru rezultāts tiek analizēts un salīdzināts ar vēlamo rezultātu. Eksperimenta modeļi tiek transformēti, lietojot modelēšanas rīku Sparx Enterprise Architect, uz kā piemēra ir parādīti tipiskie koda ģenerēšanas trūkumi. Darbā ir pārbaudīta UML klases elementa notācijas ietekme uz transformācijas rezultātu un noskaidrota pietiekošā, neizmantotā un trūkstošā notācija, kas nepieciešama klases definēšanai. Tiek secināts par iegūtā programmas koda atbilstību programmēšanas valodas likumiem un informācijas zudumiem transformācijas procesā. Iegūtais rezultāts ļauj secināt par pašreizējo koda ģenerēšanas kvalitāti modelēšanas rīkos. Raksts satur UML klases elementa sastāvdaļu aprakstus, alternatīvas notācijas piemērus, autoru sagaidāmos programmas koda tekstus un transformācijas rezultātā iegūtās programmas koda struktūras. Analīzes rezultātā ir izstrādāta ieteikumu kopa, kas dod pamatu veikt modelēšanas rīku transformācijas likumu modernizāciju koda ģenerēšanas uzdevumos.

Янис Сеянс, Оксана Никифорова. Проблемы и перспективы генерации кода из диаграммы классов языка UML
В статье рассматриваются структуры элементов из диаграммы классов унифицированного языка моделирования (англ. Unified Modeling Language – UML) и их трансформация в программный код языка C#. Нотация элементов базируется на UML 2.0 стандарте, также учитываются документы суперструктуры и инфраструктуры UML 2.2, опубликованные организацией OMG (англ. Object Management Group) в феврале этого года. Основываясь на семантике элементов и синтаксисе программного языка C#, авторы рассуждают о предполагаемом результате трансформации и условиях, которые должен учитывать генератор кода. Для анализа трансформации созданы экспериментальные модели, результат которых сравнивается с ожидаемым. Для трансформации моделей используется средство моделирования Sparx Enterprise Architect, на основе которого показаны типичные недостатки генерации кода. В статье проверяется влияние элементов диаграммы классов UML на результат трансформации достаточной, неиспользованной и отсутствующей нотации для определения системных структур. Делается вывод о соответствии полученного программного кода с правилами программного языка, а также о потере информации в процессе трансформации. Полученный результат даёт возможность делать вывод о качестве генераторов кода в средствах моделирования. Статья содержит описание элементов диаграммы классов UML, примеры альтернативных нотаций, фрагменты программного кода, ожидаемого автором работы, а также структуры программного кода, полученного посредством трансформации. В результате анализа разработан набор рекомендаций, который может служить основой для модернизации задач генератора кода в средствах моделирования систем.