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

Role of UML Class Diagram in Object-Oriented Software Development

Oksana Nikiforova[1], Janis Sejans[2], Antons Cernickins[3], [1-3]*Riga Technical University*

***Abstract* – UML is an industrial standard for object-oriented software specification which offers a notation for class modeling during object oriented software development. Since the UML class diagram is a so-called "bridge" between software specification at the user side and software realization at the developer side, it requires strong guidelines for identification of class objects from the problem domain and notational conventions for modeling of the class diagram for its further usage in system coding. This paper presents a discussion on problematic stages and possible element transformations into software components. Several conclusions are drawn on potential usage of the class diagram in industry.**

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

I. Introduction

The increasing role of modeling in software system development promotes a methodology, mostly represented by OMG solution for system abstraction, modeling, development, and reuse—Model Driven Architecture (MDA) [1]. The key component of system modeling, which underlies the principles of MDA—Unified Modeling Language (UML)—is used to define several kinds of diagrams, their elements and notation [2]. In fact, UML diagrams should be considered as a way of describing the system from various perspectives: whereas a static diagram is used to represent the structure of the system, dynamic diagrams describe its behavior.

The main goal of MDA is to provide the ability of automated transformations from platform independent models into platform-specific source code. However, due to problems with the definition of system dynamic aspects, as well as their translation into code components, this goal has not yet been achieved [3]. Nevertheless, the description of static elements alone would provide a good starting point for system development and its further refinement with dynamic aspects. This representation defined as a UML class diagram, as well as the study on possible options for generation of software components are the objects of the present research.

The class diagram, being the most common in modeling object-oriented systems [2], is used to model the static design view of a system. According to MDA [4], the automatic transition from class diagram into platform-specific software components is done by performing a model transformation, where model elements and parameters are mapped to corresponding elements and parameters in the software code.

Since published an article on a renovation of the idea of model application during software development and automatic code generation [5], the industry has still been waiting for ways to apply these ideas in software projects. This would increase productivity, while maintaining the appropriate level of software quality. Nevertheless, previous forecasts, that MDA will cover the whole area as a tsunami in next ten years (proclaimed at the European Conference of MDA in 2006), the actual impact of MDA on software development has not changed.

The authors of this paper propose to investigate the central component of model driven software development, which is the UML class diagram. Two factors are established as limitations of practical usage of the UML class diagram during software development:

1) Software developers do not invest enough effort in a formal definition of class diagram elements from the problem domain, and a class diagram is developed based on hints, human intuition and previous experience working with class diagrams. In fact, some commercial industries find that too much modeling is cumbersome and slows down productivity [6]. "For such projects, it makes sense to use UML as a sketch and have your model contain some architectural diagrams and a few class and sequence diagrams to illustrate key points" [7];

2) A survey of UML practitioners [8] shows that class diagrams are not fully used for further software development, either for code generation or documentation. The results of this report show differences in several dimensions of UML diagram usage in software development projects including the purposes for which they were used and the roles of clients/users in their creation and approval. Hence class diagram has lost the role it could play in software development – i.e. to serve as a bridge between system specification at the user side and software components at the developer side [9].

The goal of this paper is to investigate the level of class diagram usability in software development and to try to answer the following questions:

1) where is the lack of realization and application of model driven ideas in software development projects;

2) why the software industry does not apply all the ideas of MDA at high level of competence;

3) finally, why the industry is not "covered" with MDA support tools.

The paper is structured as follows. Results of the authors' research on UML class diagram usage in software development projects are discussed in Section 2. To advance practical usage of the class diagram during software development, we need a clear set of elements of the class diagram and solutions for their derivation from the problem

domain. Thus, Section 3 gives a brief review of several techniques and solutions for development of the class diagram in a more or less formal manner. Software components required at the software implementation level and several theoretical assumptions of code generation abilities are described in Section 4. Section 5 demonstrates an example of code generation from a class diagram using the Eclipse platform. In conclusion the authors refer to problems stated in the introduction and discuss questions which have been dealt with and are still open for discussion.

II. Several Issues on UML Class Diagram Usage in Object-Oriented Software Development

Regardless of what software life cycle is used, there are three main activities in software development: analysis (in conjunction with the requirement definition), design and implementation (with testing). In each of these activities, UML class diagrams are used differently.

During the analysis, while collecting information on the problem domain, the class diagram is viewed from the conceptual perspective, thus it is called a "conceptual class diagram". The diagram is used as a problem domain dictionary (potential classes) and it contains least specific notation. The reason is to facilitate communication with the customer, who is not familiar with the UML notation [10].

In the design process class diagrams are supplemented with technical details, thus more precision and more notation is used. Classes are populated with attributes and methods, as well as different kinds of relations – "structural class diagrams", which represent overall system structure (architecture). Depending on the software development process, a UML class diagram may contain a sufficiently detailed design – blueprint, while in an iterative process it can still be a general system structure – sketch.

Finally, in the implementation process, class diagrams can be used to generate system basic structure (skeleton) code. Given that the class diagram represents the structure of the system, the notation does not provide behavior of the method, however by assigning a state chart or activity diagram from a behavioral diagram group, it is possible to provide information about a body of the method, which shows system dynamics. Similarly, class diagrams can be used for system documentation. It is not necessary to reflect the whole system structure, but for example, only an individual part of the system [10]. One way or another class diagrams represent the static structure of the system. They capture domain units, resources with which the system operates, but not the operation of the system dynamics [11]. They are also abstractions from any particular system implementation - programming language syntax.

A UML class diagram serves as a primary artifact during object oriented software development. During the evolution of programming technologies and software development process in general, the current idea of basing software development on models is becoming more and more popular. According to the idea of Model Driven Architecture [1], the class diagram is a central component for representation of a solution domain in a platform independent manner and serves as a basis for generation of platform specific details that are required for further generation of a software code.

A class diagram describes the static structure of classes in the system and relationships between those entities. This way, a class diagram represents the structure of the system, as the summarized essence, the base for system operation, but not the system dynamics itself.

The key element in the class diagram is "class". The other elements are different types of relationships between classes, such as aggregation, composition or dependency. To find the usability of class diagram elements, the authors conducted a study on the UML class diagram usage in industry (in 2007) [12]. The results are summarized in Table 1.

TABLE I

Element usage based on respondents' experience

	Usable (often)	**Irregular use**	**Used in context**	**Unused (rare)**
Class element	Class stereotype	Attribute stereotype		Method stereotype
				Stereotype icon
	Attribute and method type	Method visibility	Attribute visibility	Package visibility (~) for attribute and method
			Default value for attribute and method	Tagged value
	Method parameter direction		Multiplicity for attribute and method parameter	
	Abstract method	Static method		Static attribute
				Derived attribute
				Constraint on attribute and method
Relationships	Generalization		Generalization constraint	Generalization discriminator and set name
	Composition and aggregation		Inner and outer attribute	Composite structure (composite and aggregate attribute)
	Association with defined and undefined navigability	Association name, roles, read direction, constraints	Bi-directional, one-way navigation and non-navigable association	
	Multiplicity		Dependency relationship	Dependency relationship stereotype
			Realization relationship	
Specific class	Abstract class	Active class	Association class and N-ary association	Class with qualifier
			Template class	
		Interface class	Provided and required interface Ports	Internal class structure
			XOR constraint	Powertype generalization
	Note for additional information		User-defined compartment	

The survey was sent to a number of existing software development companies in Latvia, with an aim to clarify what UML class diagram elements they were typically using in the projects. The survey showed a subsequent failure: not all companies used UML tools which could allow the diagram creator to use the wide range of notation included in the survey. This means that UML tools, as such, limit the usability of UML notation. For example, MS Visio does not provide the required interface, internal class structure (parts, ports). At the same time, there are UML tools, which can handle all notation used in the survey, like Visual Paradigm for UML.

Main elements of the class diagram marked with most common use, are: generalization, aggregation, composition and association (with defined and undefined navigability). In addition, multiplicity is marked as an important part of all these relationships. Association roles, name, direction of reading and constraint are used irregularly. In the class element, the usable parts are: type for attribute and method, method parameter direction and class stereotype, which in fact is very important for code generation.

For many elements respondents gave completely opposite assessments, thus the same element was assessed both with 10 and 0. This fact suggests difference in the level of detail used in diagrams, as well as possibly different contexts (e.g. programming in C++, Java or creating object databases) and the diagram main purpose. These elements include: template class, provided/required interface, visibility for method and attribute, default value, association class, stereotyped realization and dependency relationship, bi-directional navigation and generalization constraints. Rarely used elements are: class with internal structure, powertype notation, package visibility, stereotyped attribute and method, despite the fact, that the latter one is important for code generation.

Usually developers start modeling a class diagram directly before writing an application code, but still they construct the initial architecture defined in terms of the class diagram manually by reading the requirement specification.

Our conclusion about the usage is that first of all UML class diagram mainly is used only on a high abstraction level or as a sketch model, if it is used at all. Some respondents replied "we are not using UML at all". Secondly there are no cases with real code generation from the class diagram. We can also conclude this from the pure usage of UML 2.0 element notation invention which is straightforward for code generation, like the template class, stereotyped relationships, attributes and methods as well as tagged values and constraints. Of course there are two different views on the class diagram, i.e. visible and detailed, which are diverse in terms of the model information level.

The problem presented with two different kinds of models – a user oriented model and a developer oriented model, can be solved by generation of the class diagram from the initial knowledge about business and its complete usage for generation of software components.

In order to properly implement the usage of a class diagram, we need knowledge about:

1. Retrieving the information from the problem domain description and deriving a class diagram for further use. This information should be simultaneously complete and consistent, see current research in Chapter 3.

2. How and what software components we can get from a class diagram via transformation into code. Based on this knowledge we can define transformation rules as well as have the possibility of reengineering, see current research in Chapter 4.

Only having both components defined gives the possibility to build a bridge between user oriented models and developer oriented models of a system. In this way a class diagram helps to implement software similar to the initial description of system processes [13].

III. DEFINING A CLASS DIAGRAM FROM THE PROBLEM DOMAIN

The conceptual idea of a class diagram has been in use for a long time. Several software development methodologies and techniques used to identify classes in problem area or in initial models have been proposed since then. In fact, the approach proposed by James Rumbaugh in 1991 for Object Modeling Technique [14] still is considered one of the best approaches for identification of classes at the system domain level, with real-world operations on the domain objects and state diagrams showing the life stories of domain objects.

A. Use-Case Based Class Definition

Ivar Jacobson [15] together with Grady Booch and James Rumbaugh offered to use the definition of use-cases as a basis for software development [16]. Several other investigations also have been initiated in this direction [17], [18], [19]. A general schema of definition of a class diagram in the use-case driven approach is shown in Fig. 1 [20].

The use-case oriented approach is based on an effort to define use-cases and users of the system, as well as to describe the usage of the system with detailed scenarios that provide a basis for object interaction and sharing of responsibilities among domain classes (Fig. 1). A more comprehensive analysis of object communication results is found in the definition of class stereotypes.

However, software developers often ignore the "use-case driven", making limited or even no use of either use-case diagrams or textual use-case descriptions [8]. In fact, organizations use different tools for business process analysis and, therefore, have complete and consistent models of their organizational structure, responsibilities of the employers, business processes and the structure of documentation workflows—in other words, well-structured initial business knowledge [21].

Therefore, the class diagram may be based on the initial business knowledge (if it is formal enough).

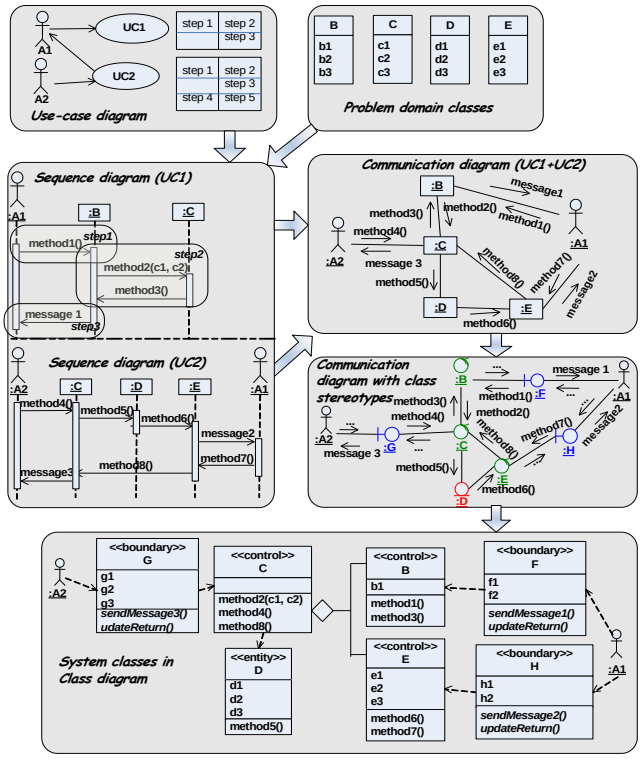

Fig. 1. Class diagram development, based on definition of use-cases [20]

So far, another group of software developers prefers to use business process modeling on the initial stages of system analysis [22], [23]. Process-oriented developers may also prefer to use use-cases, however, in this case identification of use-cases is performed in a much more formal way.

B. Data-driven System Modeling

Although only two main concepts are considered during the analysis of the problem domain—the process and data—several data-oriented approaches can be applied (e.g., such as [24]). Entity-relationship modeling (ERM) is a semiformal data-oriented technique for specifying software systems. It has been widely used for over 30 years for specifying databases [25]. Here, the developers work in correspondence with the definition of data structure since operations with data are of less importance. Of course, operations are needed to access the data, as well as the database itself which should be organized so as to minimize access time. Nevertheless, the operations performed on the data are less significant. Moreover, the manner the software is being developed is not object-oriented, also, the role of the class diagram here is secondary.

C. Two-Hemisphere Model-Driven Class Definition

In general, the concatenation of data (concept) model with the process diagram can be used to identify classes, their attributes, relationships with other classes and even more—the operations of classes. The idea of common consideration of both models is known; however, this usage in an object-oriented approach was not widely discussed. [26] proposes the way the classes and their object operations can be defined based on a two-hemisphere model [20], which essence is two interrelated models:

1) a business process model—describes the processes of the developed system;

2) a concept model—describes objects and their interaction during system work.

Two-hemisphere model-driven approach (Fig. 2) [26] proposes to start the process of software development based on the two-hemisphere problem domain model, where one model reflects functional (procedural) aspects of the business and the software system, and another model reflects the corresponding concept structures. The co-existence and interrelatedness of the models enables knowledge transfer from one model to another, as well as the utilization of particular knowledge completeness and consistency checks [20]. Then elements of the two-hemisphere model are transformed into elements of the UML class diagram, using an intermediate model and analysis of object interaction [13].

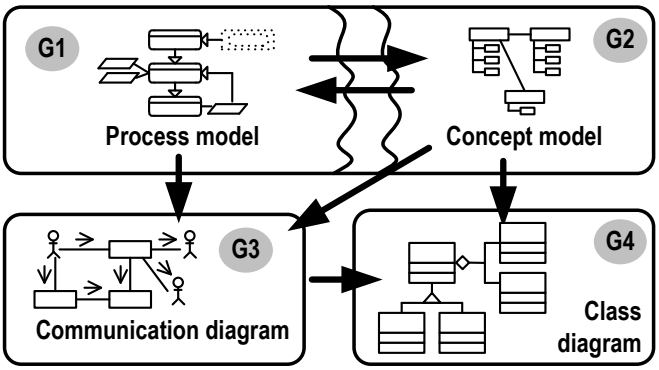

Fig. 2. Transformations from two-hemisphere model into class diagram in two-hemisphere model driven approach [26]

D. Other Techniques

Another attempt to increase the formalization level of the class diagram development is the usage of the so-called formal languages [27]. Formal languages were developed for an unambiguous system specification. Nevertheless formal languages have an important defect – only specialists are able to understand system specifications written in formal languages.

Formal languages are based on mathematics. Business specialists usually have difficulties with them. OCL [28], UML profiles [29] and executable UML [30] are some of the modeling solutions that could solve this problem. On the other hand manual transformations that are understandable for the business specialist do not support formal transformation of models at all. [31] describes the results of a survey about different approaches used for transformation of system requirements into system design and implementation. The survey shows the result of analysis on different approaches to transformation of the problem domain description into the UML class diagram during the last 10 years, published in four digital libraries (IEEEXplore, ACM, Science Direct, Springerlink) (see Table II).

TABLE II
SURVEY ON APPROACHES TO UML CLASS DIAGRAM CONSTRUCTION

Type of approach	Amount of papers for exact type of artifact
Type of problem domain description	Software 43 / Business 29
Structure of problem domain description	Standard model 25 / Non-standard model 25 / Template 1 / Structured natural language 17 / Natural language 10 / Other 3
Type of models of problem domain	Structural 9 / Behavioral 39 / Functional 6 / Other 2
Transformations provided	Yes 58 / No 14
Transformations level	Endogenous (the same language and abstraction level of source and target model) 9 / Exogenous (different) 52
Transformation type	Standard transformation level 7 / non-standard 46
Transformation automation level	Automatic 27 / Interactive (i.e. Semi-automatic) 11 / Manual 22
Tool support	Yes 25 / None 46
Type of validation	Survey 1 / Case Study 32 / Experiment 2 / None 37
Approach scope	Academic 53 / Industry 20

The survey states that there exist enough approaches with different types of solutions for the generation of a UML class diagram.

What is more, considering the class diagram construction, we can say that we have quite enough techniques for derivation of elements of the class diagram from the problem domain. The analysis of possibilities to use class diagram elements for further generation of software components is discussed in the next section.

IV. ANALYSIS OF SOFTWARE COMPONENT GENERATION

The implementation level of abstraction of the software system presented in the form of the class diagram serves as input for the code generation tool. The actual level of abstraction of programming languages has grown from the physical machine level of the first and second generation languages to the abstract machine level of the third and fourth generation languages. [32] states that one objective in using a fourth generation language is a shorter code, and, hence, quicker development and easier maintenance. The use of code generators takes three goals even further, in that the programmers have to provide fewer details to a code generator than they provide to an interpreter or compiler. Therefore it is expected that the use of code generators will increase productivity of the software system development.

A. Generation of Class Specification at the Level of Console Application

To complete the task of code generation we first need to clarify what component types are developed during system implementation. These types of components give the capability of searching for the corresponding components of a class diagram. In general, a console application of such a system can consist of classes, their definition and realization, relationships among classes, classes visibility etc. All the components required for such an application are already defined by main statements of object-oriented philosophy, the main book we can mention is [33].

So far we can see that different solutions are offered for making the process of the class diagram development more suitable, more formal, or even more "user-friendly".

The main technique here is to look for objects with the same structure (attributes) and behavior (methods) and to group them together into classes. The object is a class instance, which at the defined state performs defined operations. At the moment of object creation, the defined attribute values are assigned to an object, and another object can call it to perform the defined method. Class transformation into the C# programming language is shown in Fig. 3, where the example of the UML class is shown on the left side of Fig. 3 and the correspondent code generated from the class specification is presented on the right side of Fig. 3.

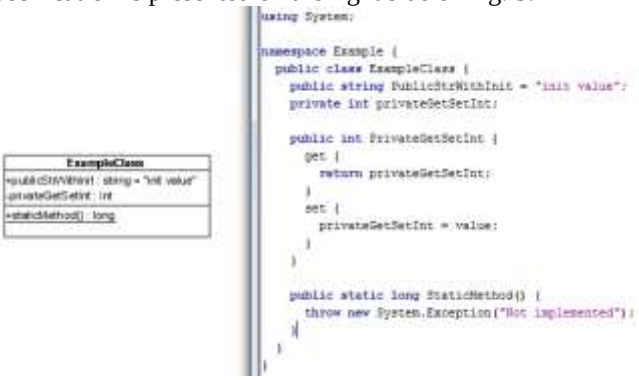

Fig. 3. Example class in both modeling and programming language

The UML class diagram offers different types of associations between classes: these serve as a basis for the generation of several other statements of object-oriented philosophy (e.g., class visibility, generalization, aggregation, usage, dependency, etc.).

All the main components of object-oriented paradigms at the console level of system abstraction are being realized since Booch, Rumbaugh and Jacobson made efforts to create code generation tools in the mid 90-ties. The Rational Rose CASE tool was created at that time and has been evolutionarily developed with IBM brand tools as its successor. A lot of open source tools have been developed since that time. However the

authors based on several experiments with code generation can conclude that despite many papers devoted to the MDA and a lot of theoretical statements about the likelihood of model transformation, in fact, the currently available tools out of the box show very poor results. Since they don't take into account target programming language syntax and models, that are not specially adapted, the tool can easily generate program code that even does not compile. In other words, current transformation is a primitive information transfer from model to code, instead of making additional decisions at the model-building and generation process stages, which as a result could at least match syntactically and semantically a proper result code.

B. Generation of Software Components for Windows/Forms Applications

The development of business-oriented software systems with user-friendly GUI interface is more complicated. In this case, the generation of a Windows/Forms application from the class diagram is considered. In general, Windows/Forms applications rely on three-layer architecture [19]: presentation, logic, and data layers (Fig. 4).

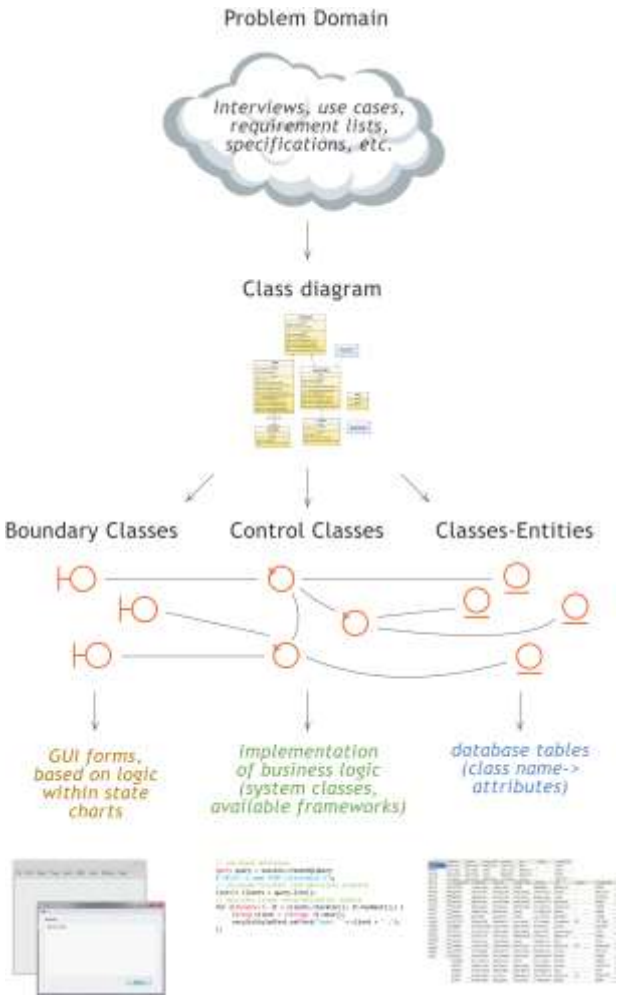

Fig. 4 The three-layer architecture

By separating the application into layers it is possible to modify each layer independently and do the technological updates for each of them without touching other layers, for example the application logic.

Presentation is the topmost level of the application, which is used to show output results from logic level. This is the interaction level between user and application, in the client-server environment the presentation level appears on the client side. Currently there are many frameworks, which support three layer architecture and where the presentation layer can be modified based on the target operating system or using style sheets in case of web environment. As mentioned, the output design can be changed without touching the other layers. From the MDA and code generation point of view it could be possible to generate a base form for each class or even concatenate related (linked) classes into one form even if the relation between classes is one to many, because it is possible to show many-side classes as a table. There are no problems to correctly guess output control for a class attribute, the decision may be based on the attribute type, for example Boolean is a check box, String is an edit box, related class is a list box etc. Also from class definition it is possible to correctly address which attributes should be visible and which are used in the background, using attribute get/set flags. If the model is enriched with a sequence or communication diagram which contains an ordered message flow in objects life, it could be possible to use this information in presentation layer by ordering input fields or even opening forms in a provided sequence. Similarly, given that the presentation level fields may be related to each other, affecting each other's output, this link could be determined from the derived class attributes and the contained fields in the derivation formula. Events which can be called during the form processing, can be labeled with appropriate stereotype which can provide visible separation of other class methods and can be transformed differently from other methods, for example with an additional windows handle parameter.

However we still believe that currently some GUI creator tools should be used at the presentation level to modify and adapt the result design, because generation can target and transform the classes thru GUI templates, which should be configured anyway.

The logic layer contains application functionality and business rules. All methods and functions are executed in this layer. At this moment class behavior transformation into executable code is the MDA weakest point, because it is not so easy to express algorithms in models. Describing class behavior UML suggests using activity, sequence, communication and state machine diagrams, so the class diagram contains method definition, but the body is expressed in the mentioned diagrams. This means that we have a choice between two options for describing class behavior in the model.

We can use the mentioned diagrams to show the object message flow, which will result in so-called "functional block" diagram difficult to read and mixing a low level code with an abstract platform independent model. Or we can use

UML defined OCL [28] and Action languages to formally describe functionality by writing a pseudo-code. There are also solutions based on UML Profile extensions such as the UML profile for EJB [3], but the problem still remains: an abstraction level is too low. It does not represent the problem as a collection of classes, attributes and relationships, instead, the result is composed of entity beans, session beans and Java classes. Needless to say, such a low level abstraction model almost coincides with the source code and transformations of this kind of model are simply straightforward, since there are no changes in the abstraction level.

However the authors believe that the fundamental problem is the lack of abstract functional components. Thus to describe class behavior it would be necessary to use pre-defined functional components, so it would be necessary to mix UML model abstraction with concrete programming languages. As an example the authors remind of the simple square root function, no one cares about what low-level algorithm implementation is used to calculate the square root. The programmer just calls ***sqrt(x)*** function and gets a result. The same analogy should be used for abstract behavior. The modeler just uses some named component by giving a class attribute as a parameter or uses stereotypes and tagged values and at the same time this component is mapped to a platform specific code. In this way a manageable and transformable model is obtained, which describes class behavior.

The data layer keeps data neutral and independent from application servers or business logic. It contains the required mapping for storing objects into a data source. To create data tables it is already possible to generate SQL scripts from a class diagram, which can be executed to build the required database instance. Similarly, it is possible to generate XML files with the correct class structure. For example, the Sparx Enterprise Architect [34] allows generating a DDL model from a PIM model, providing all the necessary transformation rules. Thus this is quite similar to generating a class definition into a target programming language.

In this way we can assume, theoretically, that all the necessary components for software system generation and the basis for such transformations rules can be defined and have to be realized by tools. Thus, the impact of the generated code on the set of software components, required to be developed, can be valuable and really powerful.

V. CURRENT FACILITIES OF SOURCE CODE GENERATION FROM THE UML CLASS DIAGRAM

The Eclipse platform together with Eclipse Modeling Framework (EMF) was selected to examine the most current facilities of source code generation, as well as to find out how model-driven approaches like MDA perform in real-life application development. In short, Eclipse is a universal tool platform and an open extensible integrated development environment (IDE) [35]. In turn, EMF extends The Eclipse platform with a solid basis for application development using modeling and code generation facilities [35].

From the MDA perspective, EMF should be considered as a framework for platform-independent and platform-specific (i.e., Java) layers. EMF utilizes the concept of the class diagram, at the same time extending it. In general, EMF has two models: the first is a meta-model, which describes the structure of the model, while the second serves as the actual implementation of it (i.e., is the instance of meta-model).

EMF uses XMI [36] to persist the model definition. The EMF meta-model definition can be defined based on [37]: XMI document, Java annotations, UML and XML Schema.

Once the EMF meta-model is specified, the generation of corresponding Java implementation classes from this model becomes possible. The source code from EMF is generated with the intention for further modifications. That is why it looks clean and documented right "out of the box" [37].

In fact, with EMF the data model explicitly enhances the visibility and extendibility of the model [37]. It also provides change notification functionality to the model in case of changes in the model happen. The EMF helps to program interfaces instead of classes. Also, it is possible to regenerate the Java source code from the model at any point of time.

The EMF Project and EMF Model wizards provide a way for defining an EMF model from UML [37]. In general, Eclipse EMF supports various model formats. However, EMF also provides additional support for IBM/Rational Software Architect (.mdl files). The reason is that RSA was used to "bootstrap" the implementation of EMF itself. Nevertheless, it is possible to create a UML model within Eclipse (via UML2 Tools project or any other UML plug-in).

The EMF model is based on two meta-models: the Ecore and the Genmodel model [35]. While the former contains information about defined classes, the latter contains additional information for code generation.

The generated source code consists of three packages [35]:

1) Model package contains interfaces and the Factory to create Java classes;
2) Model implementation package contains the concrete implementation of the interfaces defined in the model (i.e., classes);
3) Model utility package contains the AdapterFactory.

The central Factory has methods for creating all defined objects. Interfaces and their corresponding implementations contain getter and setter (if allowed) methods for each attribute. Each interface extends the base interface EObject, which together with its corresponding implementation class EObjectImpl provides a lightweight base class that lets the generated interfaces and classes participate in the EMF notification and persistence frameworks.

In order to run the generated application as a console application, the main method definition should be considered. Furthermore, if visual GUI development for such application is necessary, then one of the corresponding GUI design plug-ins for Eclipse should be considered.

While the development process of Eclipse and EMF is fairly convenient and clear, there always is a room for improvement. First of all, the current release of EMF is not final, meaning that there is still a lack of stability. In order to avoid problems with UML model import, models should contain all primitive types used to define the attributes of

model elements (at least, this is true for UML2 Tools Eclipse plug-in). As soon as an UML model is imported, the relationships among classes in UML class diagrams should be reassigned manually. This can be achieved by initializing an Ecore diagram. In fact, the Ecore diagram can be considered as an EMF representation of a UML class diagram (Fig. 5).

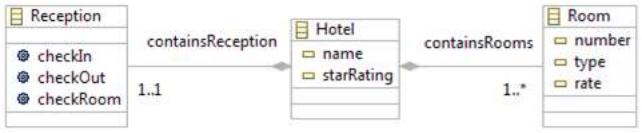

Fig. 5. An example of Ecore diagram

Eclipse EMF also lacks the option to define the data layer (like Sparx Systems Enterprise Architect). Currently, the use of additional plug-ins for modeling entity-relationship diagrams should be considered in order to generate the DDL of the database schema. However, this means that additional time is needed to develop an application, as the ER diagram should be defined from scratch.

Nevertheless, the use of other technologies such as Hibernate [38] or Teneo [39] would help to eliminate this problem by providing a framework for mapping an object-oriented domain model to a traditional relational database. The use of Hibernate and Teneo (which actually is a Hibernate representation in EMF) in itself also is a bit different.

VI. Conclusion

Abstraction is the process, where the core principles from a set of facts or statements are extracted and distilled. In turn, a model is an abstraction of something in the real world, representing a particular set of properties. There are two main reasons why developers should create a model [24]:
1) for better understanding of a process or an object by identifying and explaining its key characteristics;
2) for documenting the ideas that developers need to remember, as well as to make those ideas clear to others.

In fact, OMG's initiative—Model Driven Architecture—offers the third reason for using the models in software development [40]: models are the basis for further code generation. Moreover, the UML class diagram plays the central role in promoting this vision in the industry.

In this article, the authors investigated the current facilities of the source code generation from a UML class diagram, analyzing it from various perspectives. These perspectives include the modeling of a class diagram from initial business information, as well as concerns about the usage of a class diagram for generation of software components.

One of the contributions of the paper is a description of the state of the art in the area of UML class modeling. This paper analyzes the usage of a class diagram in software development. It looks at bigger issues – why MDE is not used in projects, why MDA is not applied in a competent way on projects, and why industry does not have good MDA tools.

The discussion surrounds the question why developers do not spend enough time developing good UML class diagrams, where one has a lot of approaches for formal construction of them.

One of the reasons stated is the assumption that the problem is not in the construction of a class diagram, but rarely occurs in code generators. The paper summarizes a representative bibliography assembled over the last 20 years in the area of object-oriented modeling analysis and approaches to the creation of a UML class diagram. The correspondent scientific literature also includes the MDA/MDD inception as well.

The main conclusion is that we have quite enough means to construct a class diagram. At least at the theoretical level we have quite enough transformations ready for solving the task of code generation during system implementation, but in practice all these means are not sufficiently supported by modern CASE tools at the sufficient level. The aspect of code generation and the results of the analysis of the quality of the code generated by modeling tools are discussed in the authors' second paper included in this issue [41]. These results clarify the question why software developers don't want to use all the facilities of a UML class diagram.

The main reason is that even if a class diagram were developed in a formal way and contained complete and consistent presentation of the problem domain, still software developers would not be able to fully use it for further software development due to weaknesses in code generation tools, because they don't support all the required transformations into code components to fulfill all the requirements of MDA. More developers and developer companies should be involved in the development of such technologies as EMF. The investment will pay off in terms of reduced amount of time spent on other projects.

Of course, the industrial companies have to meet standards of capability and maturity to be able to use all the principles and ideas of the most current facilities. However, without investment in something new and revolutionary there would be no progress in the current state of the art.

But on the other hand the lack of available powerful tools supporting all the aspects discussed in the paper can also be regarded as the most determinative and disincentive factor, which hinders valuable MDA/MDD ideas being adopted by industry. Since the renovation of the idea of model application during software development at the beginning of 21st century we still are at the same stage. This can raise doubts about the solvability of the problem.

We can discuss an analogy between code generators and automatic language translators. Even if we have a condition in both dictionaries, where the word of one language has one and only one interpretation in the other language, we can encounter a problem similar to one typical for poetry translations. All the words are translated in the correct position and sequence, but the translated text doesn't rhyme or have meaning. The same analogy can be drawn in programming language, all the code operators would be at their required places, but the program code as a whole doesn't "sound" and doesn't operate as it should.

Acknowledgments. The research presented in the paper is supported by the research grant No. FLPP-2009/10 of Riga Technical University "Development of Conceptual Model for Transition from Traditional Software Development into MDA-Oriented." The research presented in the paper partly is supported by Grant of Latvian Council of Science No. 09.1245 "Methods, models and tools for developing and governance of agile information systems".

REFERENCES

[1] "OMG Model Driven Architecture", [Online]. Available: http://www.omg.org/mda [Accessed: Sept. 24, 2010].
[2] "OMG Unified Modeling Language", [Online]. Available: http://www.uml.org [Accessed: Sept. 24, 2010].
[3] O. Pastor and J.C. Molina, *Model-Driven Architecture in Practice: A Software Production Enviroment Based on Conceptual Modeling*, Springer, 2007, pp. 302.
[4] A. Kleppe, J. Warmer and W. Bast, *MDA Explained : The Model Driven Architecture – Practise and Promise*, Addison Wesley, 2003.
[5] J. Siegel, "Developing in OMG's Model-Driven Architecture," in *OMG document omg/01-12-01* 2001. [Online]. Available: http://www.omg.org/mda/papers.htm [Accessed: Sept. 24, 2010].
[6] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2n[d] Ed., Addison-Wesley Prof., 1999.
[7] R. Miles and K. Hamilton, *Learning UML 2.0*, 1st Edition, O'Reilly Media, 2006.
[8] B. Dobing and J. Parsons, *Dimensions of UML Diagram Use: A Survey of Practitioners*, IGI Global, 2008.
[9] A. Burton-Jones and P. Meso, "Conceptualizing systems for understanding: An empirical test of decomposition principles in object-oriented analysis," *Information Systems Research*, 2006, pp. 101—114.
[10] M. Fowler, *UML Distilled: a brief guide to the standard object modeling language*, 3-rd edition, Addison-Wesley Professional, 2003. – 208 lpp.
[11] B. Unhelkar, *Verification and Validation for quality of UML 2.0 models*, New Jersey: John Wiley-Interscience, 2005, pp. 313.
[12] 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.
[13] O. Nikiforova, "Two Hemisphere Model Driven Approach for Generation of UML Class Diagram in the Context of MDA," in Huzar, Z., Madeyski, L. (eds.) e-Informatica Software Engineering Journal, vol. 3, issue 1, *Wrocław University of Technology, Institute of Applied Informatics, Oficyna Wydawnicza Politechniki Wrocławskiej*, Wrocław, Poland, 2009, pp. 59—72.
[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object Oriented Modelling and Design*, Englewood Cliffs: Prentice-Hall, New Jersey, 1991.
[15] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Professional, 1992.
[16] I. Jacobson, G. Booch and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 2002.
[17] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, New Jersey, 2000.
[18] T. Quatrany, *Visual Modeling with Rational Rose 2000 and UML*, 2nd Edition, Addison-Wesley, 2000.
[19] J. W. Satzinger, R. B. Jackson and S. D. Burd, *Object-Oriented Analysis and Design with the Unified Process*, Thomson Course Technology, 2005.
[20] O. Nikiforova, "Object Interaction as a Central Component of Object-Oriented System Analysis," International Conference „Evaluation of Novel Approaches to Software Engineering" (ENASE 2010), Proceedings of the 2nd International Workshop „Model Driven Architecture and Modeling Theory Driven Development" (MDA&MTDD 2010), Osis J., Nikiforova O. (Eds.), Greece, Athens, July 2010, SciTePress, Portugal, pp. 3-12.
[21] O. Nikiforova and M. Kirikova, "Two-Hemisphere Model Driven Approach: Engineering Based Software Development," in: 16th International Conference Advanced Information Systems Engineering. Persson A., Stirna J. (Eds.), LNCS 3084, Springer-Verlag Berlin Heidelberg, 2004, pp.219–233.
[22] M. Havey, *Essential Business Process Modeling*, O'Reilly Media, 2005.
[23] J. Jeston and J. Nelis, *Business Process Management*, 2nd Edition, Practical Guidelines to Successful Implementations, Butterworth-Heinemann , 2008.
[24] J. Toby, S. S. Teorey, T. N. Lightstone and H. V. Jagadish, "Database Modeling and Design: Logical Design," 4th Edition, in The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, 2005.
[25] P. Chen, *The entity relationship model – towards a unified view of data*, ACM Trans. Database Systems, 1, 1976, pp. 9—36.
[26] 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.
[27] A. B. Webber, *Formal Language: A Practical Introduction*, Franklin, Beedle & Associates, 2008.
[28] "OCL Specification", [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?ptc03-10-14.pdf [Accessed: Okt. 7, 2010].
[29] Ambler S.W. *Approaches to Agile Model Driven Development (AMDD)* Available: http://www.agilemodeling.com/essays/amddApproaches.htm#Manual
[30] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Boston, MA: Addison-Wesley, 2002.
[31] G. Loniewski, E. Insfran, S. Abrahao, *A Systematic Review of the Use of Requirements Techniques in Model-Driven Development*, D.C. Petriu, N. Rouguette, O. Haugen (Eds.) the Proceedings of the 13th Conference, MODELS 2010, Model Driven Engineering Languages and Systems, Part II, Oslo, Norway, pp. 213—227
[32] S. R. Schach, *Object-Oriented & Classical Software Engineering*, 7th Edition, McGraw-hill International Edition, 2007.
[33] G. Booch, *Object-Oriented Design with Applications*, 2nd Edition, Englewood City, California, 1994.
[34] "Sparx Enterprise Architect", [Online]. Available: http://www.sparxsystems.com.au/products/ea/index.html [Accessed: Sept. 19, 2010].
[35] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks, *Eclipse Modeling Framework*, 2nd Edition, Addison-Wesley, 2008.
[36] "Information Technology—XML Metadata Interchange (XMI)", International Standard, ISO/IEC 19503, First Edition, 2005.
[37] "Eclipse Modeling Framework (EMF)—Tutorial", [Online]. Available: http://www.vogellade/articles/EclipseEMF/article.html [Accessed: Sept. 19, 2010].
[38] "Hibernate—jBoss Community", [Online]. Available: http://www.hibernate.org [Accessed: Sept. 19, 2010].
[39] "Teneo", [Online]. Available: http://wiki.eclipse.org/teneo [Accessed: Sept. 19, 2010].
[40] D. Gasevic, D. Djuric and V. Devedzic, *Model Driven Engineering and Ontology Development*, 2nd Edition, Springer, 2009.
[41] J. Sejans, O. Nikiforova, *Problems and Perspectives of Code Generation from UML Class Diagram*, The Scientific Journal of Riga Technical University, Series Computer Science – Applied Computer Systems, 2011 (accepted for publication)

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.

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. His master thesis was awarded by Verner fon Siemens as a best research in 2010.

Anton Cernickins was born in 1985, died in 2010. Earned a degree of Bachelor of Engineering Sciences in 2007, a degree of Master of Engineering Sciences in 2009—both at Riga Technical University, Latvia. He was a doctoral student at the Institute of Applied Computer Systems, Riga Technical University.
He was a researcher at the Department of Applied Computer Science, RTU.

Field of interest: computer science. Special interests: modeling, model-driven development.

Oksana Ņikiforova, Jānis Sējāns, Antons Čerņičkins. UML klašu diagrammas loma objektorientētā programmatūras izstrādē
UML ir industriālais standarts objektorientētā programmatūras izstrādē, kas piedāvā dažādu sistēmas aspektu modelēšanas notāciju. Viens no sistēmas modelēšanas uzdevumiem UML notācijā ir kalpot par „tiltu" starp programmatūras sistēmas specificēšanu lietotāja pusē un programmatūras realizāciju programmētāja pusē. Šī uzdevuma risināšanai UML valodā ir jābūt definētām stingrām prasībām diagrammu elementu identificēšanai un sistēmas modeļa izstrādei, kur galveno lomu „spēlē" UML klašu diagrammas izstrāde. Klašu diagramma tiek veidota, apkopojot informāciju par izstrādājamo programmatūras sistēmu no lietotāja puses, un kalpo par pamatu programmatūras komponentu izstrādei. Pašlaik pastiprināta uzmanība ir pievērsta klašu diagrammas elementu automatizētai transformācijai koda fragmentos. Šis raksts fokusējas uz klašu diagrammas lietošanu programmatūras izstrādē no diviem aspektiem, gan aprakstot dažādus paņēmienus klašu diagrammas izstrādes metodēs no sākotnējas informācijas par sistēmu, gan arī aprakstot dažas perspektīvas koda ģenerēšanas uzdevumā. Rakstā ir secināts par to, ka eksistē pietiekami daudz pieeju, lai izstrādātu UML klašu diagrammu pilnīgu un nepretrunīgu, un ar formālām transformācijām no dažādiem problēmvides apraksta veidiem. Taču koda ģenerēšanas jomā joprojām pastāv problēmas, kā iegūt strādājošas programmatūras komponentes. Tas ir iemesls, kādēļ klašu diagrammas joprojām tiek lietotas tikai dokumentācijai. Pat, ja klašu diagramma tiks izstrādāta formālā ceļa, tik un tā, mūsdienās to vēl nav iespējams lietot koda ģenerēšanai. Patreizējā klašu diagrammas versija nesatur pietiekamu informāciju, kuru varētu "tiešā" veidā transformēt programmatūras komponentos. Koda ģenerēšanas rīki nepilda visas modeļvadāmās programmatūras izstrādes prasības, līdz ar to, pieaug rīku standartizācijas nepieciešamība. Nobeigumā tiek diskutēts par esošajām problēmām klašu diagrammas lietošanā un perspektīvām plašākai klašu diagrammas lietošanai programmatūras izstrādes projektos.

Оксана Никифорова, Янис Сеянс, Антон Черничкин. Роль диаграммы классов языка UML в разработке объектно-ориентированного программного обеспечения
UML является индустриальным стандартом для разработки систем программного обеспечения, использующим объектно-ориентированную технологию. Одной из задач моделирования системы, используя нотацию UML, является обеспечение «моста» между спецификацией системы на стороне заказчика и реализацией системы на стороне разработчика. Для решения этой задачи в языке UML должны быть определены строгие требования к идентификации элементов диаграмм и разработке модели системы, где главную роль играет диаграмма классов. Диаграмма классов строится, обобщая информацию, полученную от заказчика, и должна служить основанием для реализации системы. В последнее время большое внимание уделяется решению проблемы автоматической генерации кода из диаграммы классов. В статье рассматриваются два аспекта использования диаграммы классов в разработке программного обеспечения: с одной стороны - приемы конструирования диаграммы на базе начальной информации о системе, а с другой - перспективы генерации кода из диаграммы классов. В статье дается подтверждение того, что за 20 лет существования диаграммы классов, разработано достаточно методов, приемов и техник для того, чтобы считать, что проблема формального построения диаграммы классов решена. А вот по части генерации кода существуют проблемы, связанные с автоматическим получением работающих компонентов системы программного обеспечения. И это является одной из причин, почему диаграмма классов используется только для документации, а в полном объеме своих возможностей в процессе разработки программного обеспечения не используется, даже если разработана формальным образом. На данный момент диаграмма классов не содержит достаточный синтаксис для того, чтобы описать все возможные и необходимые элементы, которые дадут возможность генерировать полноценный программный код. И CASE средства, которые существуют в поддержку генерации кода на данном этапе, не соответствуют всем требованиям для разработки управляемой моделями системы, таким образом, возрастает необходимость стандартизировать такие CASE средства. В заключение авторы ведут дискуссию о существующих проблемах в использовании диаграммы классов языка UML и перспективах расширения области использования диаграммы классов в проектах по разработке программного обеспечения.