Начальное обучение проектированию компьютерных программ на языке UML

Введение. Проектирование является ключевым процессом жизненного цикла разработки компьютерных программ и программных систем, в котором действительно проявляется творческий потенциал программиста-разработчика и закладываются базовые характеристики (как положительные, так и, может случиться, отрицательные) разрабатываемого программного продукта, сервиса или системы. Технологии автоматизации и другие виды компьютерной поддержки проектирования возникли одновременно с индустрией программного обеспечения. С начала текущего века получило известность направление MDA (Model Driven Architecture), в основе которого лежит понимание проектирования как процесса последовательной трансформации моделей разрабатываемой программы на языке UML или на языках, родственных UML. Не став доминирующим, направление MDA тем не менее успешно развивается в контексте несколько более широкого направления MDE (Model Driven Engineering). Свободное владение языком UML и понимание его роли в MDE становится условием успешного входа профессионального программиста в мир инженерии программных систем. Этим определяется актуальность повышенного внимания к методам преподавания и особенностям освоения языка UML и, более широко, к обучению в профессиональной школе технологиям проектирования программ различного назначения и высокой сложности.

Цель статьи – предложить методику и инструментальные средства для решения проблем начального периода освоения языка UML и получения учащимися современных компетенций в области проектирования программ.

Материалы и методы. Исследование основывается на университетских учебных планах, конспектах лекций по программной инженерии, отчётах по лабораторным и курсовым работам, протоколах экспериментальных и показательных запусков разработанных программ, научных публикациях, включающих педагогическую и инженерную периодику, на Интернет-ресурсах, таких как сайты обществ ACM и OMG, и других. Использовались следующие исследовательские методы: критическая оценка собственного опыта проектирования и опыта проектирования, отражённого в студенческих работах, теоретическое концептуальное построение модели проектирования программ, включая модели сценариев проектирования, экспериментальная реализация разработанных алгоритмов.

Результаты исследования. Предложена когнитивная модель проектирования как одного из процессов разработки компьютерных программ, отражающая поведение проектировщика в части развёртывания программных проектов. С помощью этой модели сформулированы возможные сценарии оценочного и консультативного вмешательства в процесс проектирования, разработана первоначальная архитектура программного средства для такого рода вмешательства и основные алгоритмы, позволяющие в оперативном режиме анализировать и анализировать, отражённого в студенческих работах, теоретическое концептуальное построение модели проектирования программ, включая модели сценариев проектирования, экспериментальная реализация разработанных алгоритмов.

Заключение. Результаты исследования позволяют формировать среду для комфортного и эффективного проектирования компьютерных программ на языке UML путём внедрения предложенной аналитической и консультативной поддержки процесса проектирования. Предложенные алгоритмы поддержки и реализующие их программные средства ориентированы на применение в условиях начального обучения проектированию по программам высшей школы и среднего профессионального образования. Эти алгоритмы способствуют углублённому освоению языка UML и формированию современных навыков проектирования компьютерных программ.

Ключевые слова: программная инженерия, компьютерная программа, модель программы, трансформация модели, пользовательский интерфейс

Initial training in design of computer programs using UML

Introduction. Design is commonly acknowledged as a key process in the life cycle of computer programs and software-intensive systems. The process efficaciously reveals creative capabilities of the author programmer and predetermines the basic advantages as well as shortcomings of the resulting program product, service or system. Design automation and other kinds of its support have been emerged simultaneously with onset of the software industry. From the beginning of the century the MDA (Model Driven Architecture) approach grew famous, offering to comprehend a design of a program as a process of sequential transformation of the program's models, represented in UML or in other languages related to UML. Without becoming dominant, the MDA approach successfully evolves inside of a broader approach called MDE (Model Driven Engineering). Mastering UML and understanding of its role in MDE appears to be a prerequisite for a seamless entering of a professional programmer into domain of engineering of program systems. This provision justifies an increased attention to methods of UML teaching and to peculiarities of UML learning, and, moreover, to the professional education in the area of programs and program systems design technologies.

This paper is aimed to draw attentions to challenges of the initial training in UML and tackle the challenges contributing a methodology and tools for sake of fostering the quality of education in program design.

Materials and methods. The research is based on the university syllabi, the author’s lecture notes on software engineering, reports on laboratory works and course papers, protocols of experimental and sample runs of the developed programs, scientific publications, including pedagogical and engineering periodicals, conference materials, Internet resources, such as sites of the ACM, the OMG and others.

The following research methods were used: system analysis of the own experience in program design as well as experience remarkable in student’s works; theoretical conceptual modelling of the program design including modelling of design scenarios; a trial implementation of the developed algorithms.

Research results. A conceptual cognitive model of design as of a process during program development is proposed, highlighting the designer’s behavior when dealing with project artefacts. Feasible scenarios of analytic and advisory intervention into the design process are specified, using the model. A sample architecture of a program tool suitable for such an intervention is proposed as well as algorithms that enable online critical evaluation of the project's current state, prompting feasible tracks of the project's evolvement, revealing designer’s ideas and pointing some UML misuse. The algorithms are built in a program tool named Procrust that implements the proposed architecture. Trials of Procrust show that it basically fits to primary training of software design using UML and, in particular, to a training in the MDA/MDE methodology.

Conclusion. The research results enable to set up an environment for comfortable and efficient program design using UML due to employing the proposed analytic and advisory support of the design process. Algorithms of the support and supporting program tools are targeted at initial design training on curricula of higher school and secondary vocational education. The algorithms foster in-depth study of UML and acquiring of advanced software design skills.

Keywords: software engineering (SWE), computer program, program model, model transformation, Object management group (OMG), model driven architecture (MDA), user interface (UI)

For Reference:
INTRODUCTION

Design is an obligatory process in the development of any computer program. It can proceed implicitly hidden in the developer’s cognition or explicitly, that is manifesting itself by production of some artifact, usually of a document, not being yet the program under development, but carrying preliminary solutions necessary or helpful for the subsequent program implementation. The preliminary solutions are valued as a reminder for their author or a message to the contributors of the development. Nowadays it is commonly accepted that the side-products of the development are handled as a model or a series of models of a target program that will be created at last. Throughout this issue “model” stands for a model of the program being designed.

One of the widespread languages used for presentation of models is the Unified Modelling Language (UML).

The first UML specification (UML 0.8) was published in 1995; its revision UML 1.1 was adopted in 1997 as an OMG standard [1]. The revision UML 2.5.1 from December 2017 [2] manages to be stable and relevant up-to-date.

The UML language specifications do not clearly define how a user might set up the processes of deploying the language’s resources in the course of modeling. The first famous systematic method of UML application was, apparently, RUP (Rational Unified Process) [3]. Industrial application of UML grew associated with the MDA (Model Driven Architecture) phenomenon.


The OMG claimed [5, 6] that its mission includes providing standard modelling solutions of different levels using UML and other standards associated with UML, these were CWM, OLAP, MOF, XMI.

The OMG generously promised in the aforementioned documents to “generate a lot of the code and reduce the need for hand programming by an order of magnitude, and in some restricted cases we will be able to generate all of the code”; and “in a mature MDA environment, code generation will be substantial or, perhaps in some cases, even complete”.

Notions “Platform Independent Model” (PIM) and “Platform Specific Model” (PSM) play a substantial role in the MDA specifications. A PIM provides formal specifications of the system’s structure and functions abstracting away technical details, and “a PSM is expressed in terms of the specification model of the target platform”, where “platform” stands for a software infrastructure implemented with a specific software technology (CORBA, Unix, Windows or another platform) with respect to the corresponding hardware.

The OMG itself started a long-term course aimed at collecting of an open set of domain specifications, that is PIM (metamodels) for prominent areas of technology and science [7]. But OMG stood not alone. A project of a metamodel for the development of ontologies [8] may be mentioned as an example of MDA deployment independent from OMG.

The software engineering community supported the MDA initiative by multiple attempts to deploy the MDA principles and tools in projects, though other trends in the software
engineering were live: generative programming, domain specific languages, model-integrated computing, generic model management software factories [9; 10]. UML was neither the first nor the last language specially invented for modelling computer programs. Moreover, the need of languages dedicated for program modelling is reduced at the cost of writing models in the form of program blueprints immediately in programming languages.

Nevertheless, the prominent IBM has shown its adherence to MDA [11]. Later reviews and scientific publications indicated a notable interest to the development of tools supporting MDA processes. To the same topic Brian Henderson-Sellers [12] acknowledged: “UML has gone from a relatively basic descriptive tool (i.e., a tool that serves to document software systems) to a sophisticated prescriptive one – that is, a tool that can be used to specify, analyze and implement complex software systems”.

J. Bézivin [13] remarked that MDA as an initiative of the OMG be “a particular variant of a new global trend called MDE (Model Driven Engineering)”. As opposed to the MDA model, MDE model may be formalized by any formal language, not only by UML. MDE is thoroughly discussed by D.S. Schmidt [14] observing it from the perspective of the CASE (Computer Aided System Engineering).

MDA has defined [6] the notion “model” as a representation of “the business, domain, software, hardware, environment, and other domain-specific aspects of a system”. MDA model should be unambiguously paired with a definition of a modeling language. In particular, a UML model is paired with UML. With this in mind the authors will for the rest of the paper mostly consider MDA and MDE (shortly MDA/MDE) powered by use of UML.

Professional software engineering community recurringly assesses the applicability of UML, MDA and MDE. Under the pressure of a criticism this technological bundle withstood multiple revisions. The most notable among them are: UML 2.0 in July 2005, UML 2.4.1 in July 2011, UML 2.5.1 in December 2017 [2], MDA Guide rev. 2.0 [6] published in 2014. It may be stated that UML, MDA and MDE have reached a stable state. Further development seems to proceed as attempts of extending their methodical and conceptual boundaries. These are: attempts of M. Méré et al. [15] to apply formal verification methods to UML models, application of a machine learning to forecast of design operation by J. Di Rocco et al. [16] and detecting emergent behaviors in scenario-based specifications of programs by M. Jahan et al. [17].

Eventually, UML taken separately may be judged as having a plausible success story. The versions of the UML current in 2005 and 2012 were adopted as ISO/IEC standards [18; 19]. An impressive list of successful UML applications is presented on the OMG site [https://www.uml.org/uml_success_stories]. UML is highlighted in widespread monographs, in particular, in the monograph of C. Larman [3]. The practical application scale of UML may be estimated by the fact of inclusion into the high school curricula [20]. Their implementation endows bachelors in the software engineering with primary comprehension of UML.

But the life path of UML in MDE/MDA turned out to be not free from challenges. R.B. France et al. [21] giving a well weighted analysis of UML 2.0 in favor of the MDD (Model Driven Development) pointed out some pitfalls of the UML version. J. Hutchinson [22] asserted in 2014: “Firstly, domain specific languages (DSL) are far more prevalent than expected. UML is not yet universally accepted as the modeling language of choice ... more than ten years after the OMG brought out its MDA specification, there remains a lack of consensus on the best “language and tool”.

Along with constructive criticism stimulating improvement of UML and MDA/MDE a cruel criticism emerged, that not only made claims to several features and characteristics
of the UML but compromised its usefulness and right for life at the hole. Such a criticism is admitted even by a founder of UML I. Jacobson [23]. Similar reproaches come permanently from the dawn of the UML [24], UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts [22] till last year [25].

Authors tackled this rather contradictive image of UML in MDA/MDE, explored the roots of the contradictions and tried to propose a solution mitigating the challenges of UML in MDA/MDE. The solution is based on the idea of a guiding intervention into processes of initial design training with a special analytic and recommending tool (AR Tool) that simulates at the learners a sense of participation in the large-scale advanced MDA/MDE design and fosters education of appropriate skills.

The issue refers major review and surveys justifying the proposed solution and contributes a conceptual cognitive model of the design (in program development) that highlights the designer’s behavior with regard to the choice of project transformation sequence as well as of time moments when the AR Tool interventions are initiated. In the context of the model, a set of scenarios of intervention into the design process are specified.

The AR Tool is specified by a PIM that maps its sample architecture and core algorithms. During the AR Tool intervention into a design process, the algorithms assess the achieved state of the design, prompt feasible tracks of the design evolvement, reveal designer’s ideas and point out some UML misuse.

One of the AR Tool’s implementations is represented by a program tool Procrust. being result of transformations from the PIM of the AR Tool into a PSM and further into the Procrust's source code. Trials of Procrust show its applicability for primary teaching in design of programs using UML and, in particular, for training of the MDA approach.

**MATERIALS, METHODS AND INSTRUMENTS**

The research is based on the university syllabi, and the author’s lecture notes in software engineering, protocols of experimental and sample runs of the developed program Procrust, scientific publications, mainly, about UML, MDA, MDE deployment and education, Internet resources, such as sites of the ACM, OMG and others.

Following research methods were used: system analysis of the many years experience gained by one of the authors in software development and software project management in industrial as well as academia environments, including a comparative analysis of the recent scientific and engineering results relating to the program design; conceptual modelling of program design and comprehension processes on the level of human cognition and user interface of computer devices; trial implementation in Python of the proposed program tool for design and modal transformation, experimental study of the implemented tool.

The trial implementation has involved following program products:

UML editor Enterprise Architect® 15.2; IDE Microsoft Visual Studio® VS 2022; Python 3.7.0.

**RESULTS OF THE STUDY**

1. **Conceptual cognitive model of the program design**

The cognitive model is specified within two views: static and dynamic.

Static view. The static view is represented in Figure 1. Aimed to map the software design of a professional style the model represents apart of a designer’s mind a computer that
includes a Terminal and a Storage as a permanent repository of designed models. An UML editor or other editors of modelling languages are installed in the computer. At large, the model represents a designer working on an appropriate computer.

Figure 1 shows in red two additional elements: AR Tool and Resolutions inside the Terminal. The AR Tool is dedicated for the analysis and control of the design processes performed via the Editor. It is installed and executed as a separate process or a plugin of the Editor. Featuring of the AR Tool is a key item that is contributed by the current issue. The Resolutions represent recommendations elaborated by the AR Tool and displayed through the Terminal to be percepted by the designer.
The current model conveys the concept that design processes should be considered and simulated combined with comprehension processes of the program being designed. A representation of design skills is added into the procedural part of the long-term memory, nearby the representation of comprehension skills. Following the MDE vision any design process is considered as a sequence of transformations. The resulting model states are saved in the Storage. The model state of the current transformation is visible to the designer and available for editing through the Terminal.

Arrows in Figure 1 show operations that are performed by the linked components of the cognitive model. A capitalized string nearby an arrow indicates the operation’s name. Operands and results types for some operations are adjacently indicated by low case names. Names of the operations may be abbreviated.

Without any claim to simulate the cognitive processor in full, the cognitive model highlights “Constructing” and “Interpretation”, as its key operations abbreviated as Con and Int, correspondingly. The operation Con generates images of models directly in the working memory. The operation Int ponders and tries to criticize the constructed images yielding their reconstruction or replacement. The cognitive processor disposes a set of skills acquired throughout the designer’s life and applies the collected skills by calling pertinent procedures from the procedural long-term memory. The operation “Mastering” (shortly Mas) accumulates skills in the Long-term procedural memory.

Because of natural limitations of the working memory the information appearing to be excessive is swapped with declarative long-term memory by executing operations “Storing” and “Restoring” abbreviated as Sto and Res, correspondingly. The operations are executed unconsciously.

Following the skill procedures, the cognitive processor incites the Motor to execute operations “Emitting” and “Rendering” (abbreviated as Emi and Ren). The “Emitting” represents designer’s actions of controlling the Terminal and, in particular, AR Tool and Editor by means of their actual user interface. The “Rendering” results in showing of the designed model on the screen of the Editor. The operations may be translated into some class of Human-Machine-Interface operations, for example, in a notation of GOMS [26].

Controlling the Terminal from the designer’s side involves access to the Storage, represented by operations “Saving” and “Loading” (shortly Sav and Loa).

The designer perceives information from the Terminal by means of the operation “Showing” (shortly Sho), that transmits images of the designed model and control information produced by the AR Tool and the Editor into the Working memory.

Dynamic view. The view is intended to map use cases of the designer’s activity by means of a formal notation that is proposed in the issue to represent the use cases. A use case is represented by a string consisting of a series of so-called MO-sequences and definitions of macros, if any, inserted in the MO-sequences.

MO-sequence is specified by following rules.
1. Short name of an operation is MO-sequence. The short names are introduced in the description of the static view, they are: Loa, Sav, Emi, Ren, Sho, Int, Con, Cal, Mas.
2. If X is a short name, then X(type) is parameterized MO-sequence, special case of MO-sequence where type is a literal string that describes types of the operand processed by the operation in the current use case.
3. Macro is MO-sequence.
4. Remark: Macro provides substitution of one MO-sequence into another MO-sequence.
5. If A and B are MO-sequences, then A;B is MO-sequence. A;B means, that the MO-operations composing B are executed upon the MO-operations which compose A.
6. If X is MO-sequence, then X; is MO-sequence.
7. If X is MO-sequence, n is a natural number and n > 0, then Xn is a MO-expression and it means, that the execution of X is repeated n times (for example, according to the rule 4, X3 means X;X;X).
8. If X is MO-sequence, then X* is a MO-sequence and execution of X is repeated (in the meaning of the rule 4) arbitrary number of times.
9. If A and B are MO-sequences, then A||B is MO-sequence. A||B means, that the operations composing A and B are executed in an arbitrary order and/or in parallel.
10. “||” has precedence over “;”.
11. If X is MO-sequence and X is built by composition of a series of MO-sequences through “;” then [X] is MO-sequence and its constituents preserve their order if it is combined with other MO-sequence through “||”; for example, execution of [A;B]||C may really proceed in the order: A starts, C starts, A finishes, B starts, C continues execution, B finishes.
12. Space inside MO-sequence is not admitted.

Definition of macro. Macro X is an identifier specified by an expression:
X{Y}, where Y is a MO-sequence.

Basic samples:

a) Spontaneous creative activity in the designer’s mind resulting in message generation without receiving outward information may be described as:
Con;Ren;Sav(message).
The same activity augmented by a contemplation of the creation result before its saving. The contemplation is shown by including an Int (Interpretation) in the MO-sequence:
Con;Int;Ren;Sav(message).
The use case may be further developed by merging constructing and interpretation:
Con||Int;Ren;Sav(message).
If the merged activity tends to appear prolonged it may be expressed by:
[Con||Int]*;Ren;Sav(message).

b) Casual problem solution with receiving problem formulation and its onetime solution with rendering the result on the Terminal:
Sho(problem);[Int||Con]*;Ren(solution)

c) Program assimilation (comprehension) includes reading of its text, interpretation and, perhaps, mental execution, requiring the faculty to reply questions and elaborate explanations concerning the program [27]:
Sho(program);[Int*||Con*]*;Ren(explanations)]*

d) Usual design of a program starts from showing the specification of requirements for the future (conceived) program on the Terminal screen. Upon a syncretic series of interpretation and construction operations in the designer’s mind the designer decides to render the resulting model of the conceived program on the Terminal. The described actions are represented by the following MO-sequence:
Sho(requirement specification);[[Int*]||[Con*]]*;Ren(final model) //Producing final model
The MO-sequence may be taken as a body in the definition of the macro SimpleDesign: SimpleDesign { Sho(requirement specification);[[Int*]||[Con*]]*;Ren(final model)}
Further processing of the program model consists of saving it in the external Storage or constructing and saving the source code. These actions may be represented by the MO-
sequences SimpleDesign;Sav(final model) and, correspondingly, SimpleDesign;Emi(Source Code generation);Sav(source code).

Model transformation

The basic samples of MO-sequences above represent a design process consisting of only one transformation fulfilling the use case. The next model may be generated from scratch, initiating transformations of another program, or may continue the modelling process of the same program. In general case the generation process may require information saved earlier in the long-term memory or in a computer. The designer tries to restore it. Restoring from the computer is performed by operations Loa and Sho. They represent assimilation process similar to one that proceeds by program comprehension [27].

Along the issue the term “transformation” denotes a mode of design process as well as a state of the program model in course of the design. Two types of transformations are distinguished: initial transformation and forwarding.

An initial transformation is similar to SimpleDesign, but it is ended by an exteriorization of the design result through saving it in the Storage or submitted to the design participants anyway. The definition of an initial transformation Transform0 looks like:

\[
\text{Transform0}\{ \text{Sho}(\text{requirement specification});[\text{Int}^*][\text{Con}^*];\text{Ren}(\text{model}); \text{Sav}(\text{model})\}
\]

Forwarding transformation is defined as macro MidTransform:

\[
\text{MidTransform}(\text{model})\{ \text{Loa}(\text{model});\text{Sho}(\text{model});[ [\text{Sho}^*][[\text{Int}^*][[\text{Con}^*]][[\text{Ren}^*]]^*];\text{Sav}(\text{model})\}
\]

General case of program design by serial transformation performing may be expressed by the MO-expression:

\[
\text{Transform0};[\text{MidTransform}]^*
\]

Basic structural distinction between a SimpleDesign and a process of multiple transformations [MidTransform]^* both giving the same results is that the process [MidTransform]^* contains a set of additional operations Sav and Loa (Save and Load). Duration of the operations amounts to the additional time consumption of the transformational design compared with the design without transformations. The absolute values of the time consumption during a design run may be measured by catching and registering of input/output operation by means of the operating system.

2. Algorithms of analytic and advisory intervention into the design process

These algorithms are executed by the AR Tool in response to the control information emitted by the Designer. They are divided into four core groups pursuant their tasks:

control of fidelity to requirements specifications (Table 1),
control of the model completeness (Table 2),
elaboration of advices aimed at improvement of the model (Table 3),
control of the intervention depth showing which recommendations of the AR Tool were not implemented by the Designer and preparing reminding messages; the algorithms of the group will be identified as “Mx”.

Essential provision of the cognitive model is that the AR Tool is open to the extension of its repertoire of analytic and recommending algorithms. First candidates for the extension are patterns, such as GRASP or the Gang of Four patterns [27].
### Table 1

<table>
<thead>
<tr>
<th>Id</th>
<th>Name and Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>No use case without interaction diagram! Let ( X ) be a set of use cases’ names in the model, and ( Y ) be a set of names of interaction diagrams. If ( x \in X ) and ( x \notin Y ) then a warning is generated.</td>
</tr>
<tr>
<td>R2</td>
<td>No actor without life line! Let ( X ) be a set of use cases’ names in the model, and ( Y_x ) be a set of actors’ names associated with the use case ( x ). If ( Z_x ) is a set of life lines’ names in the interaction diagram named ( x ), ( x \in X ), ( y \in Y_x ), and ( y \notin Z_x ), then a warning is generated.</td>
</tr>
<tr>
<td>R3</td>
<td>Promotion of use case extension into interaction diagram. Let ( X ) be a set of use cases’ names in the model, and ( Y_x ) be a set of use cases extending the use case ( x ). If ( Z_y ) is a set of alternative blocks names in the iteration diagram and ( y \notin Z_x ), then a warning is generated.</td>
</tr>
<tr>
<td>R4</td>
<td>Copying of alternative conditions from an interactive diagram into a use case to be extended. Let ( X ) be a set of use cases’ names in the model, and ( Y_x ) be a set of extension points’ names. If ( Z_y ) is a set of conditions, ( A_x ) is a set of alternative block names and ( B_a ) is a set of conditions of the block named ( a ) and ( \forall x \in X, \forall y \in Y_x, \forall z \in Z_y, \forall a \in A_x, z \notin B_a ), then a warning is generated.</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>Id</th>
<th>Name and Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>The supported diagram should be used! Checking the availability of all types of diagrams. Let ( X ) be a set of diagrams types supported by the tool, and ( Y ) be a set of diagrams types in the model. If ( x \in X ) and ( x \notin Y ), then a warning is generated.</td>
</tr>
<tr>
<td>C2</td>
<td>No empty diagram! Let ( X ) be set of diagrams and ( Y_x ) be a set of elements in diagram ( X ). If ( x \in X ): ( Y_x = \emptyset ), then a warning is generated.</td>
</tr>
<tr>
<td>C3</td>
<td>Checking for the presence of methods in classes. Let ( X ) be a set of classes and ( Y_x ) be a set of methods in the class ( x ). If ( x \in X ) and ( Y_x = \emptyset ), then a warning generated.</td>
</tr>
<tr>
<td>C4</td>
<td>Checking the number of time lines in the interaction diagram. Let ( X ) be a set of interaction diagrams, ( Y_x ) be a set of life lines and ( Z_x ) be a set of actors by the use case carrying name of the diagram ( x ). If ( x \in X ) and power of ( Z_x &gt; 0 ) and power of ( Y_x &lt; 2 ) or power of ( Z_x = 0 ) and power of ( Y_x = 0 ), then a warning is generated.</td>
</tr>
<tr>
<td>C5</td>
<td>Checking for classes associated with a given life line. If no classes are associated with a life line in interaction diagrams of the model, a warning generated.</td>
</tr>
<tr>
<td>C6</td>
<td>No idle methods. Let ( X ) be a set of calls, ( Y_x ) is set of methods of life line’s class ( z ) connected with call ( x ). If ( x \in X ): ( x \notin Y_x ), then a warning generated.</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>Id</th>
<th>Name and Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Propagation of member names from child classes to parents. If all child classes have methods with one and the same name, an advice is generated to insert a generalized method with this name into the parent class.</td>
</tr>
<tr>
<td>F2</td>
<td>Looking for chains of interactions corresponding the given templates. All verbs in names of calls in the specified interaction diagram are collected in the order of the calls. Chains of the verbs are matched against templates gathered in the AR Tool’s repository. If there is no match or the matching chains are incomplete, then an advice is generated.</td>
</tr>
<tr>
<td>F3</td>
<td>An advice to create classes with the names of frequently encountered nouns in use cases. Nouns are collected from all use cases in the model and converted into their initial form. For nouns whose repetition rate exceeds a set threshold, for example, ( z ), a search is performed in the entire model of the same class. If there was no such match, then an advice is generated.</td>
</tr>
</tbody>
</table>
3. General methodology of design interventions

Scenarios of the proposed analytical and recommending tool interventions into the design process differ in its dynamics, depth and scope. Extreme dynamics will be achieved by intervention upon each elementary update that is permissible by the editor tool. In the case of UML editor, it may be line drawing or inserting of an inscription. The advanced editors check the compliance of the model with the actual specifications of the language used and claim the designer to rectify the model. Such a service of the editing tools reduces the load on the AR Tool and dynamics of the interventions. As long as a model preserves the compliance an intervention is initiated at the discretion of the designer, partially upon accumulation of changes along the model’s development, partially in dependance on designer’s reaction to the last recommendations. The MO-sequence expressing a forwarding transformation indicates its high dynamics by omitting of iteration sign ‘*’ at Ren operation:

\[ \text{Loa(model)}; \text{Sho(model)}; [[\text{Sho*}}]|[[\text{Int*}}]|[[\text{Con*}}]|[[\text{Ren*}}]; \text{Sav(model)} \]

Control operations involved into Editing operations are not explicitly shown in the MO-sequences considered before as far as they are hardly discernable being half-consciously executed during fluent editing. Control operations initiated deliberately may be explicitly presented in the MO-sequences and can be accounted by their evaluation.

The parameter field of the Emit let to specify the commands to be executed subsequently. The MO-sequence below expresses forwarding transformation that extends macro MidTransform involving activation of the AR Tool by a generalized command control. The following Show operation lets the designer to perceive recommendations elaborated by the AR Tool and located among the resolutions:

\[ \text{Loa(model)}; \text{Sho(model)}; \text{Emi(control)}; [[\text{Sho*}}]|[[\text{Int*}}]|[[\text{Con*}}]|[[\text{Ren*}}]; \text{Sho(notes and advice)}; [[\text{Sho*}}]|[[\text{Int*}}]|[[\text{Con*}}]|[[\text{Ren*}}]; \text{Sav(model)} \]

The depth of the intervention is meant as a degree of following the recommendations of the AR Tool. Rejection of a recommendation decreases or annihilates the depth of the intervention estimated throughout the design process. To foster the intervention depth the Reminder function is provided that logs history of the interventions and points unaccepted recommendations providing opportunity they will be accepted.

The following MO-sequence expresses a sample of forwarding transformation that reconsiders a previous transformation because of reminder from the RT Tool:

\[ \text{Loa(model)}; \text{Sho(model)}; \text{Emi(control)}; \text{Sho(reminder)}; \text{Int}; \text{Loa(previous model)}; [[\text{Sho*}}]|[[\text{Int*}}]|[[\text{Con*}}]|[[\text{Ren*}}]; \text{Sav(model)} \]

The analysis of a current model transformation provided by the AR Tool pursues two main goals: the perfection of the resulting program and attractiveness of the next transformation as the result and as process being performed by the designer. A key prerequisite of attaining the first goal is strict following the requirement specifications for the program to be designed. The careful confirmation of fidelity to the requirements in full on every step of the design is alleged to be a deposit of attaining the second one.

The results of the analysis may be prepared in different modalities: indication of faults, advice, warning, forecast. Special cases of the analysis outcome are recommendation in the form of a variant of the model or of its part and generation of the source code of the program.

Scope of the intervention depends on choice of algorithms at startup of the AR Tool. Blend of the algorithms selects a use case. The basic use case classes are Requirements specification fidelity (further – RSF), MDA simulation, Collaboration.
Hallmark of the RSF algorithms is analyzing whether the class and interaction diagrams use the vocabulary of use case diagrams of the same model. The reason is that use cases are alleged to repeat the vocabulary of the requirement specifications of the modeled program and provide thereby a fidelity to the requirements. The AR Tool is started by command \texttt{Emi(R3, C5, F3)}, where the operand refers identifiers of algorithms in Tables 1–3.

The AR Tool is started for MDA simulation by command \texttt{Emi(R1-R4, C1-C3, F2)}. The use cases may be considered as extension of the RSF use cases, concerned with fullness and consistency of the model.

Use cases of Collaboration are started by command \texttt{Emi(R1, R2, F1-F3)} and prompt possible but not obligatory directions of the design progress.

Lists of algorithms associated with the use cases are not mutually exclusive and may be extended, for example, by the algorithm Mx. A use case involving all algorithms is admitted but may slow down the performance of the AR Tool.

4. Architecture of the AR Tool

The proposed architecture of the AR Tool is represented in Figures 2-4 as Use Case, Class and Interaction diagrams. These diagrams enter in an initial PIM model of the AR Tool.

![Figure 2 Use Case diagram of an AR Tool](image)

The model was transformed into Python 3.7.0 source code constituting an application named Procrust and being executable on different operating platforms, in particular, on platforms: Intel + Windows 10+ and Intel + Ubuntu 22+.

![Figure 3 Class diagram of an AR Tool](image)
5. Implementation of a sample AR Tool

5.1 Architecture of Procrust

The developed tool Procrust is deployed as application shown in Figure 5. Except of executable file Procrust.py, other notable files are Procrust.xml containing an analyzed model and Prom_comments.txt for resolutions resulting from the analysis. File Params.txt contains custom and built-in algorithms settings (types of algorithms, their priority and description). Procrust implements the AR Tool with some limitations.

Avoiding any plug-in connection to UML editors Procrust accesses the processed model, as do similar tools, in the XMI format exported by most UML [28]. The ready-made parsers are provided, the use of which significantly expands the developer's capabilities.

Figure 4 Interaction diagram of an AR Tool and its user

Figure 5 Procrust in the file system
Procrust analyzes only three types of UML diagrams, namely: use cases, classes and system interactions. The choice of these diagram types is necessary because of the need for balance between structural and behavioral information, and it is sufficient as most scholarly projects implicate no other diagram types.

The analytic and recommender algorithms are included in Procrust as plug-ins. Params.txt file specifies names of the algorithms and its parameters. The user can add his own algorithms.

Algorithms F2 and F3 require natural language analysis, therefore the rnnmorph neural network is integrated in them, performing the analysis with an accuracy of up to 95%. The neural network was pre-trained by its creator [29] and supports various operations with data in the natural language. Two of them are used in Procrust, these are mechanism of selecting parts of speech in a sentence through a morphological analysis of the sentence and mechanism of lemmatization. Call of the mechanisms is shown in Figure 6.

```python
#setting the language to be analyzed by the neural network
predictor=RNNMorphPredictor(language="ru")
#forming a list of words from and their parameters using a neural network
words=predictor.predict(string)
#lemmatization of words by necessity
for word in words:
    if flag==True:
        words=word.normal_form.casefold()
#use of the received data in further work of algorithms
```

Figure 6 Sample of calling the neural network

Procrust provides automated source code generation by connecting a code generator as plug-in. It is connected the same way as analytic and recommending algorithms, that is through Params.txt file indicating location of the code generator, being an independent executable program.

The architecture of the tool is tied to the MVC template favoring parallel operation of the tool and the user himself. An example of recommendations prepared by Procrust as result of an analysis can be seen in Figure 7.

Figure 7 A window of Procrust showing recommendations
5.2 Approbation of Procrust

Procrust was tested in three stages: testing of all types of algorithms, self-testing (self-application), development of a software product from scratch for a student course paper.

The task of the course paper (the 2nd year of MAI) was to develop a program that provides an interaction of the user with some object (in particular, a phone). The development was carried out twice with an interval of 2 months without the use of Procrust and with it. The results match with an accuracy of 97%. However, the time and effort spent on the same job varied significantly. The results are shown in Table 4.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Without Procrust</th>
<th>With Procrust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time spent</td>
<td>4 hours 36 minutes</td>
<td>2 hours 29 minutes</td>
</tr>
<tr>
<td>Total number of transformations</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Number of large-scale transformations</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

It can be concluded that the use of Procrust has increased performance of the development process by 65%. It is important to note that Procrust provides feature that enables the designer to implement recommendations received from Procrust by clicking the “solve the problem” button. Procrust will perform the necessary changes automatically. Implementation of this feature for all recommendations will significantly boost the design performance.

THE DISCUSSION OF THE RESULTS

1. Problems of MDA/MDE and the solutions

History of the program design sketched in the issue’s introduction indicates that the UML should not be considered separately from MDA and, moreover, from MDE.

An amazing ambiguity is thereby visible. Large and, perhaps, middle scale industrial projects enjoy success due to adoption of the MDE technology by the developers-programmers. But the same technology is not acknowledged by mass of developers, without a doubt, loners and newbies among them. Clearing the cause of the ambiguity might help to find resources for better teaching and deployment of UML and MDA/MDE.

General factors of the ambiguity seem to be quite simple.

Main positive impacts of MDA deployment on design processes are known for a long, these are: promoting the understandability of models, projects inward interoperability, project reusability. The impact is distributed depending on the project specifics. Big team projects enjoy interoperability offered through sequences of models represented in UML or in domain specific languages. The interoperability has proven to be a decisive factor of success for large and mid-scale endeavors in software engineering. So, if a technology provides the interoperability, its unfavorable side-effects may be excused. Such an experience might explain the positive side of the MDE reputation.

Low-scale projects are less sensitive with regard to the extent of interoperability. The applicability of MDA to low-scale projects depends on the balance between the suspected
negative factor, from one side, and the gain of understandability of models and other possible positive impacts, from the other side.

Report of J. Hutchinson et al. [22] contributes to considering of the aforementioned factors. The report is based on a survey, that engaged more than 400 respondents experienced in practical MDE applying and over 1000 years’ total experience in software development. Developers’ opinion gained by the participation in some mid-range successful projects was also included. The investigations published in 2014 were conducted in 2009-2010, a decade upon the MDA proclamation. The questions verbatim and the corresponding answers partially shorten are presented in Table 5 and Table 6.

Table 5

<table>
<thead>
<tr>
<th>Id</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What are models used for?</td>
<td>The answers are presented in Table 6.</td>
</tr>
<tr>
<td>2</td>
<td>Which modeling languages are used?</td>
<td>UML – 85,5%</td>
</tr>
<tr>
<td>3</td>
<td>Which diagrams are used?</td>
<td>Class diagrams</td>
</tr>
<tr>
<td>4</td>
<td>Benefits of code generation?</td>
<td>The ability to automatically generate code is considered an important part of the productivity gains they achieve (&gt;75%).</td>
</tr>
<tr>
<td>5</td>
<td>Making changes to the model or to the code?</td>
<td>Modifying the models rather than the code is the preferred approach.</td>
</tr>
<tr>
<td>6</td>
<td>Is UML too complex? Is UML powerful enough for your needs?</td>
<td>Almost half of our respondents still believe that UML is too complex and almost a third believe that, despite this complexity, it is not yet powerful enough.</td>
</tr>
<tr>
<td>7</td>
<td>Does MDE promote understanding?</td>
<td>The answers are contradictory: poor comprehension does not harm the understanding if the usage of the language is consciously simplified or results in misunderstanding if the limitations get violated.</td>
</tr>
<tr>
<td>8</td>
<td>Is tooling a barrier to the use of MDE?</td>
<td>25% said they definitely are, with another 30% saying they probably are.</td>
</tr>
<tr>
<td>9</td>
<td>Impact on personal experience, if MDE hindered or helped a variety of personal processes ...</td>
<td>8%–13% of respondents think that their use of MDE actively “hinders a lot” their productivity, enjoyment, creativity and problem solving.</td>
</tr>
</tbody>
</table>

Full-scale analysis of the tables goes beyond the scope of the issue, which is limited at this point to clearing of the overall distribution of UML and MDA acceptance.

Table 6

<table>
<thead>
<tr>
<th>Id</th>
<th>Answer to the question 1 in Table 1</th>
<th>Percentage of negative answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Use of models for understanding a problem at an abstract level</td>
<td>4,8</td>
</tr>
<tr>
<td>2</td>
<td>Use of models for team communication</td>
<td>7,3</td>
</tr>
<tr>
<td>3</td>
<td>Use of models for capture and document design</td>
<td>9,4</td>
</tr>
<tr>
<td>4</td>
<td>Code generation</td>
<td>11,8</td>
</tr>
<tr>
<td>5</td>
<td>Use of model-to-model transformation</td>
<td>27,8</td>
</tr>
<tr>
<td>6</td>
<td>Use of domain specific languages</td>
<td>32,3</td>
</tr>
<tr>
<td>7</td>
<td>Use of models in testing</td>
<td>35,1</td>
</tr>
<tr>
<td>8</td>
<td>Model simulation/Executable models</td>
<td>39,1</td>
</tr>
</tbody>
</table>
Unfortunately, the report does not provide information sufficiently to credibly deduce correlations between different answers. Nevertheless, Table 6 highlights that the praxis under consideration falls in two groups, namely answers 1–3 and answers 5–7. Respondents of the first group who gave a positive answer used models in amount of at least primary mastering of UML or other domain specific language. Respondents of the second group who also gave a positive answer used models on the level of an overt application of MDE.

Answers to the questions 2 and 9 in Table 5 give a reason to conclude that majority of respondents (85.5%) use and tolerate UML. Therefore, the stated split into groups of respondents dully pertains to the stable UML users. Answer to the question 6 then implies, that a considerable part of the respondents among those, who use UML, is not satisfied with it apparently whether because of its complexity or due to lack of the expressive power.

Conclusion 1. Almost all respondents use a limited part of the MDE competences, exactly those, who are provided by a starting level of the UML experience. Population of designers acquainted with MDE and UML in some extent falls into 2 parts. One of the parts gains from mastering MDE and constitutes a “silent minority”. The other part perceives MDE as an overhead and blames it from time to time.

This conclusion though needing further experimental confirmation turns the attention to problems of the UML and MDA/MDE education and, broadly, to the topic of training in design.

Due to the successes in industry and an abundant coverage in the scientific literature the object-oriented modeling and, particularly, UML with MDA/MDE have penetrated in software engineering curricula [20], syllabi (for example, [30; 31]) and manuals, meeting quite appreciable negative feedback.

J. Cabot et al. [32] outline in details an experience of introducing MDA to undergraduate students. They had to produce some MDE artefacts with help of Eclipse [33] and develop a Web-application with the WebRatio tool [34]. The participants have convinced that they have got a bad opinion about MDE and will try to avoid to use it once more. It is remarkable that the tool used are not specialized for the teaching and provide a high degree of automation including code generation. As follows from the referred paper the tools worked unstable. Besides, in conditions of an intensive automation the students had few opportunities to experience MDE in its basic form, that is recollecting MDE principles and applying them through immediate model composing and transformation in UML.

D.C. Schmidt writes [14] that developers modeling in the course of program designing spare to invest any considerable efforts in it.

The authors on the base of their personal experience of teaching and learning program design in high school share the opinion. Students try to neglect UML modelling during program development and tend to pass immediately to composing of the source code.

An attentive consideration of UML and MDE teaching from a newbie’s perspective discover several causes that obstruct the teaching.

Teaching of MDE begins as any teaching from simplified training “toy” tasks. The student has no opportunity to experience the power (usefulness) of the MDE in full, including positive effects of interoperability and reusability. In this case, automation contraindicates the very sense of teaching the design being a creative occupation. Only the automatic code generation may be admissible. But the usual UML editors provide a trivial kinds of code generation hardly useful and weakly didactic.

By the manual code generation quality of the resulting program does not visibly depend on the last transformation of the UML model. The model does not look like a decisive result of developer’s efforts, undermining the designer’s motivation.
From the other side, UML does not require, with rare exceptions, formal connection between diagrams. A model may appear to be a set of diagrams connected only semantically through similarity of names assigned to the diagram elements. This disunity meets acceptance by some researches and related tools in MDA. As example, it may be mentioned: refactoring of class diagram by O.A. Derjugina [35], validating and optimizing object and class diagrams by K. K. Sergievskiy et al. [36], exploration of object and class diagrams by Kästner et al. [37], synthesis of State Machine Models N. Kahani [38], insight about class diagram using in the survey of J. Hutchinson [22].

Reducing of diagram set deprives the model of fulness. It will be hardly perceived by its author as result of a designer’s holistic endeavor, negatively effecting his/her motivation.

**Conclusion 2.** MDA as knowledge seems to be poorly scalable with regard to teaching. Expressing its power in the space of huge projects with significant transformations, it ceases to be justified in small projects and short fictitious transformations during scholarly lessons.

Highly likely, this state of affairs appears to be a sample of expectancy theory of V. Vroom [39]. According to this theory motivation \( M = E \times I \times V \), where \( E \), \( I \) and \( V \) denote expectancy, instrumentality and valence. Elaboration of a model decreases senses of instrumentality and valence, and, hence, decreases the motivation as compared with elaboration of the very targeting program without any explicit modelling.

**Conclusion 3.** There is a need of some special organization of teaching MDA hardly realizable without a special tool support.

Direct browsing in the Internet as well as perusal of notable reviews, such as the report of J. Hutchinson, considered above, survey of E. Jakumeit et al. [40] and investigation of S. Lathouwers and V. Zaytsev [41], covering two decades of MDE development let hardly find a tool dedicated to teaching of MDE.

E. Jakumeit et al. [22] have thoroughly compared 13 transformation tools selected to represent the state of the art in the worlds of model transformation and graph rewriting. It is remarkable that the tools offer the user a language or access to a set of specialized languages for operating over models and metamodels. It implies that the tools are hardly applicable to primary teaching MDE on the base of pure UML. Only exception is made by the UML-RSDS tool, via which software systems can be specified and designed using UML class diagrams, use cases, state machines and activities.

S. Lathouwers and V. Zaytsev [41] have investigated the domain of program verification tools, and presented a concise megamodel splitting the domain into 7 levels. No one of the tools attributed to the domain was explicitly dedicated for teaching.

**Conclusion 4.** The displayed analysis suffices to propose an approach to mitigate the drawbacks of current MDA education. The approach introduces a special class of tools that should:

- enable interventions into the design process assessing the flow of MDE transformations and managing it through recommendations;
- raise senses of instrumentality and valence of current transformations as intermediate goals in the design process, boosting the designer’s motivation in accordance with the Vroom’s concept;
- simulate for the designer an organizational framework complying with the last Hutchinson’s remark about importance of social and organizational prerequisites for MDE deployment [22];
- accommodate the design process to the tasks of design in general and to individual requirements of the designers.
The authors presume that the AR Tool contributed in the issue may be considered as a representative of the specified class.

2. Conceptual cognitive modelling

The model of program design introduced in this issue (Figure 1) extends the generic model of program comprehension, proposed in [27], and follows trends noticed in the review of A. Fekete, and Z. Porkoláb [42]. The extension is justified as far as any design process of a program design implies a process of comprehension of the program, at first piecemeal and as whole at last. Design involves not only inward mental but also creative real-world activity. Correspondingly, the model is enriched by mapping of the real-world tools (Terminal, Storage, AC Tool, Editor) and operations with them. The enrichment entails dividing of the model into views, these are static and dynamic. The division is made mainly for sake of the observability.

The dynamic view takes care to express an interplay of cognitive and real-world operations. A special notation is introduced that tries to address the incomplete understanding what order cognitive processes are performed in and claims to express parallelism and perhaps partially random order of the processes.

MO-sequence is the main type of expression admitted in the notation. It allows linear operation sequences solely, but it is sufficiently for description of design scenarios.

The sequences could be represented on activity UML diagrams, of course, but the formula view of the MO-sequences appears to be more concise and observable.

MO-sequence specifies boundaries of the set of tracks that are eventually left by the execution of operations implicitly as well as explicitly during design and comprehension of a program. Operations that involve hardware and software of the Terminal may be recognized by the ordinary means of the operating system and logged by the AC Tool operations of the Human-Machine-Interface. The operations of perception might be empirically investigated by technologies of gaze tracking.

Construction as a process in the development of any computer program may be simulated much like the design process differing mainly by deployment of other procedures from the procedural long-term memory.

The notation may be extended by admitting “if-then-else” clause and assigning time attributes to use cases and operations.

3. AR Tool functionality

Though no any modal transformation is the primary destination of the AR Tool, it may be positioned in the category of Direct-Manipulation Approaches within hierarchy proposed by Czarnecki [9]. AR Tool does not match definitions of the Model-To-Code Approach as far as a run of AR Tool does not inevitably entails some code generation. But if the designer approves the advice to generate the source code? The generation may be performed via Procrust, and it falls under the Visitor-Based Approaches of the Czarnecki hierarchy.

Choice of the functionality is implied by goals of intervention into design process. Analysis above highlights that the process of teaching UML and MDA may suffer if the UML diagrams appear to be disjoint. One of the important special cases ensuring consistency of a model is complying of class and interaction diagram with structural and lexical solutions that ascend to the initial requirements of the project and grow expressed in the use case diagrams. The architectural choice implemented in the AR Tool follows a series of works paying considerable attention to the mapping of requirement specification in the designed model.

So, H. Zhang et al. [43] have proposed and tested a methodology of deriving mutation operators for use case models. M. Chabot et al. [44] have provided a unified testing
framework to enable the validation of cross-disciplinary requirements. B. DeVries et al. [45] have developed Automatic Detection of Incomplete Requirements via Symbolic Analysis. M. Jahan et al. [17] have implied a machine learning approach to detecting unexpected scenarios in software requirements modeling. Implementation of last three contributions could be potentially applied as preprocessors of Procrust.

Lexical uniformity of a model confirms that it was designed responsibly and with full understanding. Automated generation of class diagrams from requirements given in natural language as in [46] ensures the conformity. Newbies should be able to provide the uniformity without automation. AR Tool takes over checking of the competence. But the designer should be aware that AR Tool analysis is performed on the level of names assigned to the elements of the analyzed model.

The scientific literature delivers numerous reports scoping tools and methods that help the designer to improve particular aspects of the designed model. For example, A. Kästner et al. [37] propose a transition to class models starting from object models. V. Besnard et al. [47] provide formal verification of some model properties expressed with Linear Temporal Logic. O. A. Deryugina [35] has proposed a program tool for refactoring of UML class diagram. The ideas implemented in the aforementioned works can enrich Procrust’s functionality in a course of its further development.

M.V. Sergievskiy and K. K. Kirpichnikova [36] have contributed to validating and optimizing UML class diagrams, following, in particular, several patterns, well-known in the software engineering. Analysis and recommendations whether a model complies with some kind of pattern do not belong to the core capabilities peculiar to the AR Tool. The authors believe that an enthusiasm with patterns might be premature for an initial UML training overshadowing basic language knowledge. But tracking of some patterns is included in the Ptocrust functionality.

Due to its checking capabilities Procrust can be attributed to the level PV1 of the megamodel proposed in [41], playing a role of the “model solver”.

Observations made by R. Jolak et al. in [48] substantiate that tool-related challenges are the most encountered and the universal tools (Matlab and Simulink) concede the specialized ones (Papyrus and PapyrusRT) in the context of software modelling. The comparison of Procrust with other MDE tools shows that Procrust accommodates features of several approaches continuing to be a peculiar case tool true to its original teaching destination.

**Future work**

Studies that can be advanced on the base of current contribution relate:

- processing OCL annotations in UML models,
- processing state diagram analysis, taking in account high degree of design automation achieved by N. Kahani generating about state diagrams [38],
- formal verification following [15],
- forecast [16] using machine learning,
- processing UML templates [49].

**CONCLUSION**

The research results enable to set up an environment for comfortable and efficient program design using UML due to employing the proposed analytic and advisory support of the design process. The support is based on a qualified intervention in the design process by a special analytic and recommender tool that introduces into the process
REFERENCES


2. OMG Unified Modelling Language (OMG UML). Version 2.5.1.(2017), 1-754. Available at: https://www.omg.org/spec/UML/2.5.1/PDF


organizational and managerial factors that lead to success or failure. *Science of Computer Programming, 89*, Part B(01), 145 – 161. DOI: 10.1016/j.scico.2013.03.017


30. Syllabus on direction 09.03.04 Software engineering. (2023). Moscow Aviation Institute MAI(NIU) 202324. Available at: https://cp.mai.ru/docs/pub/3fd010b287378c07c0eaa17a3e0971a/default/7& (accessed 27 August 2023).


Information about the authors

Nikita S. Andreyanov
(Russia, Moscow)
Master student
Moscow Aviation Institute
(National Research University)
E-mail: nik00789030127799@bk.ru
ORCID ID: 0009-0001-2465-5445

Andrej P. Gagarin
(Russia, Moscow)
Cand. Sci. (Tech.), Professor of the Department
"Computers, Systems and Networks"
Moscow Aviation Institute
(National Research University)
E-mail: gagarin_ay@outlook.com
ORCID ID: 0000-0002-0929-2834