Challenges in Preserving Intent Comprehensibility in Software

Valentino Vranić,[1] Jaroslav Porubän,[2] Michal Bystrický,[1] Tomáš Frťala,[1] Ivan Polášek,[1 3] Milan Nosáľ,[2] and Ján Lang[1]

[1] Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 2, 84216 Bratislava, Slovakia, vranic@stuba.sk, tomas.frtala@stuba.sk, michal.bystricky@stuba.sk, jan.lang@stuba.sk

[2] Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 04200 Košice, Slovakia, jaroslav.poruban@tuke.sk, milan.nosal@tuke.sk

[3] Gratex International, a. s., Galvaniho 17/C, 821 04 Bratislava, Slovakia, ipo@gratex.com

Abstract: Software is not only difficult to create, but it is also difficult to understand. Even the authors themselves in a relatively short time become unable to readily interpret their own code and to explain what intent they have followed by it. Software is being created with the goal to satisfy the needs of a customer or directly of the end users. Out of these needs comes the intent, which is relatively well understandable to all stakeholders. By using other specialized modeling techniques (typically the UML language) or in the code itself, use cases and other high-level specification and analytical artifacts in common software development almost completely dissolve. Along with dedicated initiatives to improve preserving intent comprehensibility in software, such as literate programming, intentional programming, aspect-oriented programming, or the DCI (Data, Context and Interaction) approach, this issue is a subject of contemporary research in the re-revealed area of engaging end users in software development, which has its roots in Alan Kay's vision of a personal computer programmable by end users. From the perspective of the reality of complex software system development, the existing approaches are solving the problem of losing intent comprehensibility only partially by a simplified and limited perception of the intent and do this only at the code level. This paper explores the challenges in preserving the intent comprehensibility in software. The thorough treatment of this problem requires a number of techniques and approaches to be engaged, including preserving use cases in the code, dynamic code structuring, executable intent representation using domain specific languages, advanced UML modularization, 3D rendering of UML, and representation and animation of organizational patterns.

Keywords: intent; use cases; domain specific languages; 3D rendering of UML; organizational patterns

1 Introduction

Software is not only difficult to create, but it is also difficult to understand. Even the authors themselves in a relatively short time become unable to readily interpret their own code and to explain what intent they have followed by it, i.e., what they wanted to achieve. Similar situations arise with models and in particular with more detailed, design models. This problem is being solved by introducing another artifact into software development: documentation. This brings in a further complex problem: the need to keep documentation up to date. In the case of internal documentation (comments), the traceability of the artifacts the documentation is related to has to be ensured, too. Furthermore, a considerable effort is needed to initially create the documentation, inevitably with a disputable and difficult to control quality because its consistency, as opposed to that of a program, cannot be tested by actual execution.

This problem can also be perceived in a more global manner. Software is being created with the goal to satisfy the needs of a customer or directly the needs of the end users. Out of these needs comes the intent, which is relatively well understandable to all the stakeholders. By using other specialized modeling techniques (typically the UML language) or in the code itself, use cases and other high-level specification and analytical artifacts in common software development almost completely dissolve.

Understanding the intent expressed in code has been identified as one of the key problems in software development that has a direct impact on creating programming languages and related tools [36]. This is important not only in software maintenance, but it is also related to the question of reuse: the comprehension of the intent as realized by a given component is necessary for its reuse.

Along with dedicated initiatives to improve preserving intent comprehensibility in software, such as literate programming, which subordinates code to documentation [34], intentional programming, which aimed at enabling direct creation of appropriate abstractions by the programmer [57], aspect-oriented programming, which makes possible to gather parts of the code into modules by use cases [25, 26], or the DCI (Data, Context and Interaction) approach, which enables to partially preserve use cases [14], this issue is a subject of the contemporary research in the re-revealed area of engaging end users in software development [6] that has its roots in Alan Kay's vision of a personal computer programmable by end users. From the perspective of the reality of complex software system development, the existing approaches solve the problem of losing intent comprehensibility only

partially, by a simplified and limited perception of the intent and do so only at the code level.

This paper explores the challenges in preserving intent comprehensibility in software. The thorough treatment of this problem requires a number of techniques and approaches to be engaged. Section 2 explains the dimensions of the intent in software. Sections 3–5 explore the possibilities of preserving intent comprehensibility from the perspective of each of the basic software constituents. Section 6 discusses the related work. The paper is closed by conclusions and indication of further work directions.

2 Dimensions of Intent

The ultimate form of the software is the executable *code*, which is mostly text based. The level of the comprehensibility of the intent expressed by the code determines the quality of the resulting software system: better intent comprehensibility simplifies error discovery and lessens the divergence from the functionality that the software system should have provided.

Software development is accompanied by the creation of many artifacts that as such do not contribute to its functionality and thus fairly quickly become outdated. These additional artifacts are usually conjointly referred to as documentation. A special position among these is held by *models* as non-code artifacts predominately expressed in a graphical form and used to reason about the software system being developed. As such, they can be perceived as a transient form towards the code, which is, after the code has been created, condemned to outdating. Models can also be used to generate code or other models, or they can even be executable. In any case, it is important for the intent in models to be comprehensible, too.

The originators of the intent are *people* and losing the intent in organizing people is transferred to the software being developed by these people. This is a direct consequence of Conway's law [11]:

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

This is not only the problem of the initial organization of the people. The people remain the key factor throughout software development including the maintenance phase, too. They tend to literally impersonate the software system parts they develop and the effectiveness of resolving development problems depends directly on the effectiveness of the communication among the developers. This is where agile and lean approaches, which favor face-to-face contact among people over any kind of formal communication, save significant time and resources [14].

Furthermore, the notion of intent in software is relative and it can be observed in the following dimensions:

- *Stakeholders* (intent originators): from whose perspective the intent is observed, starting with the end user and moving towards the programmer
- *Level*: at what level of construct granularity is the intent expressed, starting with the lower level constructs (e.g., conditional statements or loops), via covering constructs (e.g., methods, classes, etc.), conceptual constructs and software system parts (different levels of subsystems), up to the overall software system, including people organization
- *Expression*: how is the intent expressed, starting with an idea or mental model, via informal notes and further forms of textual description, including requirements lists and use cases, up to graphical model representations, formal specification, and, finally, the executable form itself

The entire intent space as determined by these dimensions, i.e., *stakeholders–level–expression*, is huge. With respect to the basic software constituents, three particular areas of interest can be identified therein (see Figure 1):

- With respect to code, it is reasonable to observe the intent by its representation in an executable form at the code level up to the covering constructs level
- With respect to models, the focus should be on expressing the intent by graphical models spanning from the conceptual construct level up to the software system level
- With respect to people, the most interesting is the people organization as such and people organization in projects, at which the employment of textual description and graphical models to express the intent appropriately should be explored

All three areas span throughout the whole stakeholder dimension since, in general, any stakeholder can be involved in any software artifact. This is at heart of the agile and lean approaches to software development with their concept of cross-functional roles. It may seem strange for end users to be connected with code, but it a huge number of people in the USA (four times the number of professional programmers there) reported they do programming at work [6]. With techniques that shape code according to use cases and domain specific languages, addressed in the next section, end user programming becomes even more relevant. Sections 4 and 5 address the remaining two areas of interest in exploring the intent in software.

3 Code Perspective

As we will see in this section, the intent expressed in use cases can actually survive in code (Section 3.1) with application domain abstractions supported by domain specific languages (Section 3.2), while the differences in the mental models of individual stakeholders require abandoning the fixed code structure (Section 3.3).

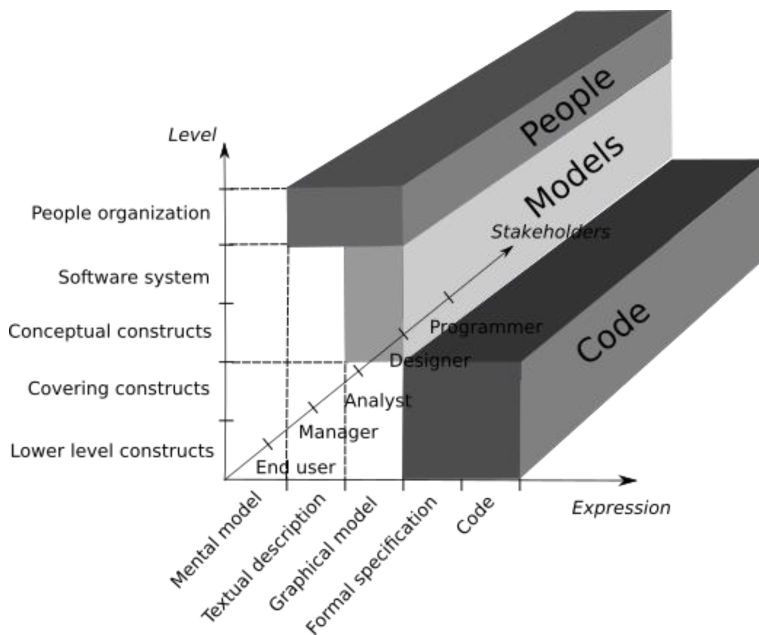

Figure 1

Areas of interest in exploring the intent in software

3.1 Preserving Use Cases

From the perspective of preserving use cases in code, there are two approaches of particular interest: aspect-oriented programming, which enables to collect the code parts into modules corresponding to use cases [25, 26], and the DCI approach [14], which enables to partially preserve use case flows, i.e., sequences of steps in

use cases—known also as *flows of events* or simply *flows*[1]—albeit they remain fragmented by roles.

In common object-oriented programming, the client or end user intent diminishes from code. The parts of use cases end up in different classes by which they are realized. The ability to affect one use case by another one without having any reference to the affecting use case in the affected one, known as the extend relationship, also lacks in common object-oriented programming.

What should be explored is how appropriate design patterns, the frameworks based on these design patterns, and preprocessing techniques can ensure not only the code to be modularized according to use cases, but also how use case flows (the actual steps) can be preserved in the code and how to achieve this in a form close to the natural language. The modification of use cases, which usually happens only in code without reflecting the changes in the use case model, would consequently be readable to stakeholders that have no programming knowledge. This would even open a possibility for these stakeholders to directly modify the program, at least at the highest level.

It is necessary to investigate the possibilities of expressing individual steps in use case flows and the possibility of their formal interpretation without fully formalizing how they are expressed. For this, formal processing of informal meaning by abstract interpretation could be considered [35]. By now, it has been demonstrated, though only in the Python programming language, how use case flows can be preserved in code [7]. The modularization of use cases using design patterns in the PHP programming language has been addressed, too [23], but this approach does not preserve use case flows.

3.2 Domain Specific Languages

Domain specific languages are being created with the goal of bringing closer the way the code is expressed to the problem domain. Programmers use problem domain notions directly in the solution, while tools communicate with them in this notional apparatus, too [42]. Domain specific languages play a key role in model driven software development. Because of this, the orientation on domain specific languages in searching for the ways to express the intent in software comes as a natural choice. Despite a growing popularity of domain specific languages, a number of questions remain unanswered in this area. These include language composition [18], effective sentence creation using projectional and hybrid editors [62], and assessment of the usability of domain specific languages [4].

[1] Sometimes *scenario* is used to denote a use case flow. This may be confusing since a scenario can stand for a particular path through a use case, which involves some or all steps of one or several use case flows.

Domain specific languages can be used to achieve a readable and executable intent representation that enables to propagate the notions from use cases, i.e., application domain, to code, i.e., solution domain. In other words, domain specific languages raise the level of abstraction of the solution domain closer to the application domain abstractions making it possible for programmers to express the solution using in the application domain fashion.

There are three promising directions in the research of preserving intent comprehensibility with domain specific languages. The first one is to come closer to the ideal case in which a domain specific language will be both the application domain language and solution domain language. A domain specific language can have textual, but also graphical (visual) concrete representations, which constitutes the grounds for the second direction of research in preserving the intent with domain specific languages: overcoming the gap between the model and the code, while retaining executability. The third direction aims at simplifying the creation of domain specific languages and their evolution, which takes place along with the evolution of the program itself (constituting a program–language coevolution).

3.3 Dynamic Code Structure

A huge impediment to observing the intent is the fixed code structure. Different stakeholders need to see code in different ways in which—from their perspective—the intent is readable. However, these cannot be based on a static code representation, but have to be modifiable as though they are the actual code representation. The challenge here is not only to design the necessary views, but also to enable creating further views directly by stakeholders.

The fixed code structure is a consequence of the economic aspects of software development. Developers are bound to choose exactly one code structure. This code structure corresponds to the mental model they have built upon their experience and knowledge, but also to the nature of the intent they have to realize. An alternative implementation that would employ some other code structure is in this case redundant and thereof economically inconvenient. Thus, the final code structure usually favors one intent that the authors considered to be the most important. That intent can be anything, including technical intents such as software efficiency, software extensibility, etc.

The programmers that join a project during the course of its realization, have to understand the existing code before they can manage to progress. In such a situation, the new programmers, as new stakeholders, have to adopt the mental model of the original programmers. This is difficult, since their own experience, knowledge, and preferences in general are only rarely close to those of the original programmers and thus constitute a mental barrier to the code comprehension. If, for example, the original programmers focused on the system efficiency, while the new programmers prefer extensibility, they might not understand many of the

decisions made by the original programmers making it difficult for them to comply with these decisions.

The problem of the limitations of static structure is in practice partially being attacked by the built-in projections in integrated development environments (IDE). Finding variable uses, which enables to follow scattering of the variable use and thus to follow the intent implemented by it, is one of these. Similarly, environments enable to follow selected intents that are not directly part of the executable code. One example is comments with the *TODO* prefix, which can be found and provided to stakeholders by the IDE in a navigable list with all the instances of a given comment.

Approaches are known that improve certain aspects in the context of the problem area of dynamic code structuring. A prominent example is a method for a faster orientation in code supported by a tool that enables to display the body of the method being called directly at the place of the call without the need of explicit navigation [16].

Intentional code views from the perspective of the architectural intents based on logical metaprogramming have been reported [41]. However, these views are not editable. Intents have been represented in a form of a graph abstraction, too [55]. Programming with so-called ghosts [8] is based on automatic creation of undefined yet entities used in the program, which are displayed in a separate, editable view. The approach has been implemented as a prototype in the form of an Eclipse plugin and as an extension to Smalltalk Pharo. Recording and automating design pattern application treated, for example, by Kajsa [27], can also be perceived as a way of expressing the intent using metadata.

4 Model Perspective

Dynamic code structuring as addressed in Section 3.2 is conceptually applicable to graphical models, too. Providing different, modifiable views instead of only one, static view is highly related to aspect-oriented modularization or to what is known as advanced modularization in general. There are some opportunities to achieve this in UML as a de facto standard in software modeling (Section 4.1). Employing the third dimension in representing software models can further improve intent comprehensibility (Section 4.2).

4.1 Advanced Modularization in UML

Despite extensive research in the area of aspect-oriented software development, expressing advanced modularization at the model level did not end up with a generally accepted approach. A very important approach in this direction is Theme

[10], which is, similarly as newer approaches such as RAM [33], based on non-UML elements.

Separation of concerns with clearly expressing their interrelation naturally contributes to intent comprehensibility. The UML diagrams typically used in practice make this possible only to a limited extent. Therefore, it is necessary to investigate how advanced elements of UML, which usually have no straightforward counterparts in code, can help in expressing intent. In this sense, composite structure models, whose important elements are roles and their collaborations, are particularly interesting. Here, it is necessary to search for the ways of expressing roles and their composition. Another UML concept that is directly interconnected with composite structure models are parameterized types.

4.2 3D Rendering of UML

Model rendering itself and creation of alternative views can also enhance intent comprehensibility by reducing its fragmentedness. The third dimension can be employed here to simultaneously display and interconnect related parts of the model. For this, a complex 3D UML rendering support for the layout of class or module layers and their relationships has to be provided. This is different than the standard package modularization, in which the relationships between package elements are not easily observed. The point is in maintaining the layers in a simulated 3D space in which it will be possible to create and observe elements and their relationships along with the relationships between the layers as such.

In this sense, the model could be structured according to use cases as intent bearers. Use cases define interaction and therefore are commonly modeled by sequence diagrams, which implicitly uncover the underlying structure necessary for the realization of use cases [26, 1]. To expose the structure directly, sequence diagrams can be easily converted into communication or object diagrams and displayed in their own layers. The class diagram automatically created out of the communication or object diagrams could be displayed in another layer making the correspondence—and their intent—of the elements of these different diagrams obvious provided their planar coordinates are preserved. The idea itself has been indicated earlier [47]. Current research efforts aim at realizing it [22]. This approach is applicable also to decoupling patterns and antipatterns in class diagrams depicted schematically in Figure 2.

Several approaches to the 3D rendering of UML diagrams have been reported. However, each of these approaches is targeting only one type of UML diagrams and none of them supports editable 3D views. X3D-UML [39] is an approach to the 3D rendering of UML state machine diagrams in movable hierarchical layers with the possibility of applying filtering. No appropriate tool for editing this view is available.

GEF3D [17, 45] is a 3D framework based on Eclipse GEF (Graphical Editing Framework) developed as an Eclipse plugin. By using this framework, existing GEF-based 2D editors can easily be embedded into 3D editors. The main idea of this framework is to use the third dimension to visualize connections among several common (2D) UML diagrams each of which is displayed in a separate layer parallel to other layers. GEF3D supports also orthogonal positioning of layers into virtual boxes [17] that makes inter-model connections clearly visible, but limits the number of layers. GEF3D views are non-editable. Moreover, the project has not been maintained since 2011.

A different way of 3D rendering of UML diagrams is based on so-called geons [9], simple geometrical forms by which humans recognize more complex objects according to Biederman's recognition-by-components theory. In UML diagrams, a different geon is assigned to each kind of model element. According to this approach, by getting used to this mapping, even complex diagram structures become readily comprehensible.

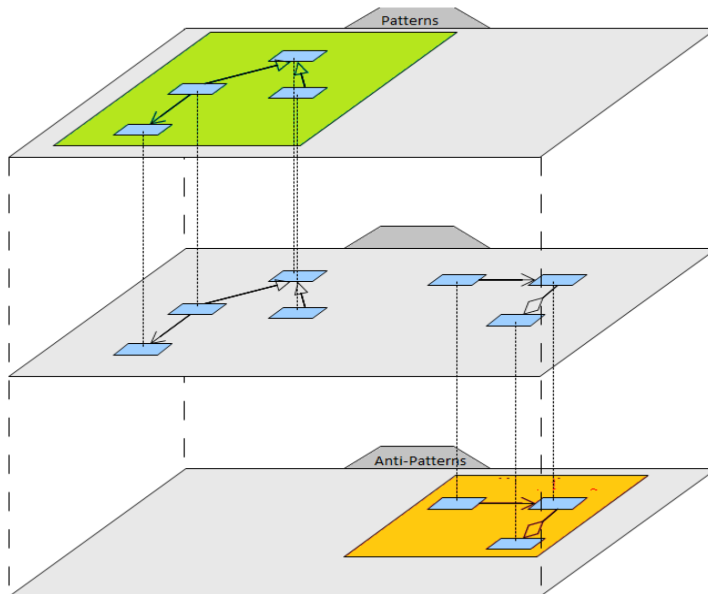

Figure 2

Decoupling patterns and ant-patterns with a layered 3D rendering of a class diagram

5 People Perspective

Appropriate ways of organizing people have been discovered in successful projects of software development and captured as organizational patterns [15]. Approaches that will enable to better understand the intent of organizational patterns individually and in combination have to be explored. One possibility is their clarification using software modeling and UML in particular, including the 3D rendering of UML discussed in Section 3.3. This approach is applicable to expressing the organization of people in areas other than software development, too.

Alternatively, organizational patterns could be modeled in a virtual world in which it would be possible for people to try the roles featured in these patterns and experience the characteristic problem situations in a simulated environment.

For example, in the Architect also Implements pattern [15], a stakeholder could play the role of a programmer who has to understand the design of a software system in order to be able to implement it. The stakeholder could also play the role of a software architect who prepares the design without a clear idea of its implementation. The stakeholder could also experience each of these roles in a positive arrangement in which the architect cooperates with programmers and contributes to the implementation.

A virtual world does not necessarily mean virtual reality. That would surely be a benefit, but the human imagination is capable of substituting this dimension when important features of the content are captured. This phenomenon is known in videogames, among which those with an interesting story tend to endure despite a simpler graphical workout. Thus, the essence is in creating the corresponding model of a typical situation solved by a given organizational pattern. In general, it is necessary to find an approach of transforming organizational patterns into such typical situations and to create a framework in which they could be readily expressed.

Agile games [37, 20] provide a possibility to experience situations in which it is possible to better understand principles, relationships, and forces acting in agile and lean approach to software development. The same approach could be applied with organizational patterns.

Differently than agile games, the envisaged representation and animation of organizational patterns in a virtual world would provide a more realistic picture of the situation. It would also take less time and would be applicable in an individual setting with no need to engage other people, nor depend on their time. A simplified representation of this idea is offered by the SimSYS environment [12]. Animating organizational patterns as text adventure games [21] promises to improve the comprehensibility of original descriptions of organizational patterns [15].

6 Related Work

The problem of preserving intent comprehensibility accompanies software development from its beginnings. In the introduction, we indicated some important historical points starting from Alan Kay's vision of personal computer programmable by end users [32] through several approaches to preserving intent in programming, namely literate programming [34], intentional programming [56], aspect-oriented programming [25, 26], and the DCI approach [14]. Preserving intent comprehensibility is a subject of research in contemporary area of engaging end users in software development [6]. Even though involving the client or end users in software development is important, too, we claim that the successful treatment of the problem of preserving intent comprehensibility has to comprise all three basic software constituents: code, models, and people.

Aspect-oriented change realization [5, 40, 64, 65] enables modular expression of a change, by which it actually contributes to a more comprehensible representation of the intent. Determining the realization type of a change that has to be applied is possible also by the multi-paradigm design with feature modeling [40], which is in its own right interesting from the perspective of preserving intent comprehensibility because it represents an effort to bridge the gap between the solution and application domain by a transformation [63].

The intent in code can be exposed by encouraging programmers to record their intent in the form of intentional comments prior to writing any code as in the design intention driven programming [38]. Such enforcement of documentation directly in the code addresses some of problems and limitations of literate programming [58].

Approaches strongly bound to a model, e.g., domain driven design [19], do not consider exposing behavior at a higher, use case level, but rather they encourage programmers to create knowledge-rich models—also called smart models—which do not evolve well [13]. As a result, the higher-level use cases become fragmented in models and they are not visible in the code. This is addressed by several approaches that preserve the intent in code at a higher level by the modularization of the code into use cases [7, 14, 25, 26].

A use case modeling metamodel can serve as a basis for reasoning about preserving use cases in the code and models [66, 67].

In applying advanced modularization for the purpose of a clearer separation of the intent in code, techniques of applying advanced modularization in established programming languages that are not being denoted as aspect-oriented [3] are of particular interest.

The problem of effective design of domain specific languages from the perspective of tool support [52] is the key to a successful adoption of domain specific languages in software development. A domain specific language is a dynamic

element in software development undergoing constant changes. This evolution of domain specific languages may involve their composition. In building domain specific languages, creating their concrete syntax out of the samples of sentences of abstract conception [53], as well as inferring domain specific languages out of user interfaces of existing software systems [1], can help significantly.

Recording the applied design patterns using annotations [56] can help in preserving intent comprehensibility in code. The method of abstracting information from the details of the format being used, targeting XML formats and annotations [43], can be applied here. Furthermore, an analysis of the need for dynamic code structure and its conceptual design has been reported [44], supported by a NetBeans module prototype that enables simple intent based code projections expressed by structured comments [54].

Using 3D space for software modeling in UML has been reported to support code refactoring and optimization by displaying patterns in a separate layer, as well as antipatterns for refactoring in another layer [46, 48, 50, 60]. Code visualization upon AST project graphs [49, 51, 61] can be used to present models. This is related not only to algorithms of positioning elements and their relationships, such as FM3 or Fruchterman-Reingold, but also to the internal conception of presenting information in code, such as authors, anitpatterns, types of classes, and similar [22].

To successfully master the graphical demands while working with UML models in a 3D space, advanced software visualization [24, 28, 30, 59] and graph visualization in general [31], as well as design of environments for visual programming [29], are necessary.

Conclusions

Up to now, preserving intent comprehensibility has been approached to in a fragmented manner without clearly understanding its relativity. We envisage an integral perception of the intent in software and in support to its comprehensibility in the whole space formed by the identified dimensions of the intent, i.e., stakeholders–level–expression, and in all three basic software constituents, i.e., code, models, and people. Our hypothesis is that it is possible to increase the comprehensibility of the intent by applying the corresponding methods conceived with respect to the dimensions in which the intent is observed:

- Having use cases as part of the code can overcome current fragmentedness of the representation of use case flows, as well as readability of their steps as such
- Domain specific languages can provide a readable and executable intent representation
- Dynamic code structuring brings in editable adaptable code views

- Advanced modularization in UML strives at employing standard yet underutilized UML elements to express the intent
- 3D rendering of UML has the potential of providing a fully editable model capable of displaying the relationship of use cases to the corresponding detailed UML model (applicable to the DCI approach, too), but also patterns and antipatterns in optimization and refactoring, aspects in aspect-oriented modeling, and alternative and parallel scenarios
- Representation and animation of organizational patterns of software development will make possible to transfer the experience of proven ways of organizing people in software development in a new form available to all stakeholders on an individual basis and at a convenient time

Acknowledgement

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12.

This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] J. Arlow and I. Neustadt. UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design. 2nd Edition, Addison-Wesley, 2005.

[2] M. Bačíková, J. Porubän, D. Lakatoš: Defining Domain Language of Graphical User Interfaces. In Proceedings of SLATE '13, 2nd Symposium on Languages, Applications and Technologies, Porto, Portugal, 2013.

[3] J. Bálik and Valentino Vranić. Symmetric Aspect-Orientation: Some Practical Consequences. In Proceedings of NEMARA 2012: International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, at AOSD 2012, Potsdam, Germany, ACM, 2012.

[4] A. Barišić, V. Amaral, M. Goulão, and B. Barroca. Quality in Use of Domain-Specific Languages: A Case Study. In Proceedings of 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, ACM, 2011.

[5] M. Bebjak, V. Vranić, and P. Dolog. Evolution of Web Applications with Aspect-Oriented Design Patterns. In Proceedings of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with ICWE 2007, Como, Italy, 2007.

[6] M. M. Burnett and B. A. Myers. 2014. Future of End-User Software Engineering: Beyond the Silos. In Proceedings of the Future of Software Engineer-

ing, FOSE 2014, 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, ACM, 2014.

[7] M. Bystrický and V. Vranić. Preserving Use Case Flows in Source Code. In Proceedings of 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015, Brno, Czech Republic, IEEE Computer Society, 2015.

[8] O. Callau and É. Tanter. Programming with Ghosts. IEEE Software, 30(1):74–80, 2013.

[9] K. Casey and C. Exton. A Java 3D Implementation of a Geon Based Visualisation Tool for UML. In Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, Kilkenny City, Ireland, 2003.

[10] S. Clarke and E. Baniassad. Aspect-Oriented Analysis and Design: The Theme Approach Addison-Wesley, 2005.

[11] M. E. Conway. How do Committees Invent?, Datamation, 14(4):28–31, April 1968.

[12] K. Cooper and C. Longstreet. Towards Model-Driven Game Engineering for Serious Educational Games: Tailored Use Cases for Game Requirements. In Proceedings of 17th International Conference on Computer Games, CGAMES, IEEE, 2012.

[13] J. O. Coplien. The DCI Architecture: Supporting the Agile Agenda. Øredev Developer Conference, 2009. https://vimeo.com/8235574

[14] J. O. Coplien and G. Bjørnvig. Lean Architecture: for Agile Software Development. Wiley, 2010.

[15] J. O. Coplien and Neil B. Harrison. Organizational Patterns of Agile Software Development. Prentice Hall, 2004.

[16] M. Desmond, M.-A. Storey, and C. Exton. Fluid Source Code Views. In Proceedings of 14th IEEE International Conference on Program Comprehension, ICPC'06, Washington, DC, USA, IEEE Computer Society, 2006.

[17] K. Duske. A Graphical Editor for the GMF Mapping Model. GEF3D Development Blog, 2010. http://gef3d.blogspot.sk/2010/01/graphical-editor-for-gmf-mapping-model.html

[18] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In Proceedings of 12th Workshop on Language Descriptions, Tools, and Applications, ser. LDTA '12, New York, NY, USA: ACM, 2012.

[19] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003. ISBN: 0-321-12521-5.

[20] J. M. Fernandes, S. M. Sousa. PlayScrum—A Card Game to Learn the Scrum Agile Method. In Proceedings of 2nd International Conference on Games and Virtual Worlds for Serious Applications, 2010.

[21] T. Frťala and V. Vranić. Animating Organizational Patterns. In Proceedings of 8th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2015, ICSE 2015 Workshop, Florence, Italy, IEEE, 2015.

[22] L. Gregorovič, I. Polášek, and B. Sobota. Software Model Creation with Multidimensional UML. In Proceedings of International Conference on Research and Practical Issues of Enterprise Information Systems, CONFENIS 2015, 23rd IFIP World Computer Congress International Conference on Research and Practical Issues of Enterprise Information Systems, LNCS, Daejeon, Korea, Springer, 2015 (accepted).

[23] J. Greppel and V. Vranić. An Opportunistic Approach to Retaining Use Cases in Object-Oriented Source Code. In Proceedings of 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015, Brno, Czech Republic, IEEE Computer Society, 2015.

[24] F. Grznár and P. Kapec. Visualizing Dynamics of Object Oriented Programs with Time Context. In Proceedings of Spring Conference on Computer Graphics, SCCG 2013, Smolenice, Slovakia, ACM, 2013.

[25] I. Jacobson. Use Cases and Aspects—Working Seamlessly Together. Journal of Object Technology, 2(4):7–28, 2003.

[26] I. Jacobson and P.-W. Ng. Aspect-Oriented Software Development with Use Cases Addison-Wesley, 2004.

[27] P. Kajsa and P. Návrat. Design Pattern Support Based on the Source Code Annotations and Feature Models. In Proceedings of 38th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'12, Springer, 2012.

[28] P. Kapec. Hypergraph-Based Software Visualization. In Proceedings of International Workshop on Computer Graphics, Computer Vision and Mathematics, GraVisMa 2009, Plzeň, Czech Republic, 2009.

[29] P. Kapec. Visual Programming Environment Based on Hypergraph Representations. In Proceedings (Part II) of ICCVG 2010, International Conference on Computer Vision and Graphics, Warsaw, Poland, LNCS 6375, Springer, 2010.

[30] P. Kapec. Visualizing Software Artifacts Using Hypergraphs. In Proceedings of Spring Conference on Computer Graphics in Cooperation, SCCG'2010, Budmerice, ACM, 2010.

[31] P. Kapec, Michal Paprčka, Adam Pažitnaj, and Vladimír Polák. Exploring 3D GPU-Accelerated Graph Visualization with Time-Traveling Virtual Cam-

era. Journal of Theoretical and Applied Computer Science (JTACS), 7(2):16–30, 2013.

[32] A. Kay. A Personal Computer for Children of All Ages. In Proceedings of the ACM Annual Conference – Volume 1 (ACM '72), ACM, 1972.

[33] J. Kienzle, W. Al Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein. Aspect-Oriented Design with Reusable Aspect Models. Transactions on Aspect-Oriented Software Development, 7:279–327, 2010.

[34] D. E. Knuth. Literate Programming. The Computer Journal, 27:97–111, 1984. http://www.literateprogramming.com/knuthweb.pdf

[35] J. Kollár. Formal Processing of Informal Meaning by Abstract Interpretation. In Proceedings of Smart Digital Futures 2014, SDF-14, Frontiers in Artificial Intelligence and Applications, Volume 262, Chania, Greece, IOS Press, 2014.

[36] T. D. LaToza and B. A. Myers. Hard-to-Answer Questions About Code. In Proceeding of Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '10, Reno, Nevada USA, ACM, 2010.

[37] T. Lynch et al. An Agile Boot Camp: Using a LEGO®-Based Active Game to Ground Agile Development Principles. In Proceedings of 2011 Frontiers in Education Conference, FIE 2011, Rapid City, SD, USA, 2011.

[38] K. Matz. Designing and Evaluating an Intention-Based Comment Enforcement Scheme for Java. MA thesis. Walton Hall, Milton Keynes, MK7 6AA, United Kingdom: The Open University, 2010.

[39] P. McIntosh. X3D-UML: User-Centred Design. Implementation and Evaluation of 3D UML Using X3D. PhD thesis, RMIT University, 2009.

[40] R. Menkyna and V. Vranić. Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling. In Proceedings of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Revised Selected Papers, LNCS 7054, Krakow, Poland, Springer, 2012.

[41] K. Mens, B. Poll, and S. González. Using Intentional Source-Code Views to Aid Software Maintenance. In Proceedings of International Conference on Software Maintenance, ICSM ’03, Washington, DC, USA, IEEE Computer Society, 2003.

[42] M. Mernik, J. Heering, A. M. Sloane. When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR), 37(4):316–344, 2005.

[43] M. Nosáľ and J. Porubän. Supporting Multiple Configuration Sources Using Abstraction. Central European Journal of Computer Science. 2(3):283–299, 2012.

[44] M. Nosáľ, J. Porubän, and M. Nosáľ. Concern-Oriented Source Code Projections In Proceedings of Federated Conference on Computer Science and Information Systems, FEDCSIS 2013, Kraków, Poland, IEEE, 2013.

[45] J. von Pilgrim and K. Duske. GEF3D: a Framework for Two-, Two-and-a-Half- and Three-Dimensional Graphical Editors. Proceedings of 4th ACM Symposium on Software visualization, Ammersee, Germany, 2008.

[46] R. Pipík and I. Polášek. Semi-Automatic Refactoring to Aspect-Oriented Platform. In Proceedings of 14th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2013, Budapest, IEEE, 2013.

[47] I. Polášek. 3D Model ako spôsob návrhu objektovej štruktúry (3D Model for Object Structure Design). In: Systémová integrace, 11(2):82–89, 2004. In Slovak.

[48] I. Polášek, P. Líška, J. Kelemen, and J. Lang. On Extended Similarity Scoring and Bit-Vector Algorithms for Design Smell Detection. In Proceedings of IEEE 16th International Conference on Intelligent Engineering Systems, INES 2012, Lisbon, Portugal, IEEE, 2012.

[49] I. Polášek, I. Ruttkay-Nedecký, P. Ruttkay-Nedecký, T. Tóth, A, Černík, and P. Dušek. Information and Knowledge Retrieval within Software Projects and their Graphical Representation for Collaborative Programming. Acta Polytechnica Hungarica, 10(2):173–192, 2013.

[50] I. Polášek, S. Snopko, and I. Kapustík. Automatic Identification of the Anti-Patterns Using the Rule-Based Approach. In Proceedings of IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics, SISY 2012, Subotica, Serbia, IEEE, 2012.

[51] I. Polášek and M. Uhlár. Extracting, Identifying and Visualisation of the Content, Users and Authors in Software Projects. Transactions on Computational Science XXI: Special Issue on Innovations in Nature-Inspired Computing and Applications, LNCS 8150, Springer Berlin Heidelberg, 2013.

[52] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek: Annotation Based Parser Generator. In: Computer Science and Information Systems, 7(2):291–307, 2010.

[53] J. Porubän, J. Kollár, M. Sabo: Abstraction of Computer Language Patterns: The Inference of Textual Notation for a DSL. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013.

[54] J. Porubän and M. Nosáľ. Leveraging Program Comprehension with Concern-Oriented Source Code Projections. In Proceedings of Slate`14, 3rd Symposium on Languages, Applications and Technologies, Bragança, Portugal, 2014.

[55] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In Proceedings of 24th International Conference on Software Engineering, ICSE ’02, Orlando, Florida, USA, ACM, 2002.

[56] M. Sabo and J. Porubän. Preserving Design Patterns Using Source Code Annotations. Journal of Computer Science and Control Systems, 2(1):53–56, 2009.

[57] C. Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. Technical report, MSR-TR-95-52, Microsoft Research, 1995.

[58] M. Smith. Towards Modern Literate Programming. Honours project report, University of Canterbury, New Zealand, 2001.

[59] M. Šperka, P. Kapec, and I. Ruttkay-Nedecký. Exploring and Understanding Software Behaviour Using Interactive 3D Visualization In Proceedings of 8th International Conference on Emerging eLearning Technologies and Applications, ICETA 2010, Stará Lesná, The High Tatras, Slovakia, 2010.

[60] M. Štolc and I. Polášek. A Visual Based Framework for the Model Refactoring Techniques. In Proceedings of 8th IEEE International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010, Herľany, Slovakia, IEEE, 2010.

[61] M. Uhlár and I. Polášek. Extracting, Identifiyng and Visualisation of the Content in Software Projects. In Proceedings of 2012 4th World Congress on Nature and Biologically Inspired Computing, NaBIC 2012, Mexico City, Mexico, IEEE, 2012.

[62] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In Proceedings of 7th International Conference on Software Language Engineering, SLE 2014, 2014.

[63] V. Vranić. Multi-Paradigm Design with Feature Modeling. Computer Science and Information Systems Journal (ComSIS), 2(1):79–102, 2005.

[64] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing Applications with Aspect-Oriented Change Realization. In Proceedings of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2008, Revised Selected Papers, LNCS 4980, Brno, Czech Republic, Springer, 2011.

[65] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog. Aspect-Oriented Change Realizations and Their Interaction. e-Informatica Software Engineering Journal, 3(1):43–58, 2009.

[66] V. Vranić and Ľ. Zelinka. A Configurable Use Case Modeling Metamodel with Superimposed Variants. Innovations in Systems and Software Engineering: A NASA Journal, 9(3):163–177, Springer, 2013.

[67] Ľ. Zelinka and V. Vranić. A Configurable UML Based Use Case Modeling Metamodel. In Proceedings of 1st Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2009, Novi Sad, Serbia, IEEE Computer Society, 2009.