DOI:10.2298/CSIS110406047V

Improving Program Comprehension by Automatic Metamodel Abstraction

Michal Vagač[1] and Ján Kollár[2]

[1] Department of Informatics, Faculty of Natural Sciences, Matej Bel University
Tajovského 40, 974 01 Banská Bystrica, Slovakia
michal.vagac@gmail.com
[2] Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Jan.Kollar@tuke.sk

Abstract. The maintenance of a software system represents an important part in its lifetime. In general, each software system is the subject of different kinds of changes. Bug fixes and a new functionality extensions are the most common reasons for a change. Usually, a change is accomplished by source code modifications. To make such a modification, correct understanding the current state of a system is required.
This paper presents the innovative approach to the simplification of program comprehension. Based on the presented method, the affected software system is analysed and metamodel for the selected feature is created. The feature represents functional aspect of a system being the subject of the analysis and change. The main benefit is that by focusing on well known (and precisely described) parts of program implementation, it is possible to create metamodel for implementation parts automatically. The level of metamodel is at a higher level of abstraction than implementation.

Keywords: Aspect-oriented programming, feature location, metalevel architectures, program comprehension, reverse engineering, software change.

1. Introduction

Software systems help us with many everyday tasks. Since none software system is perfect, sooner or later there is demand for a change. Software systems often model a real world situation. That is why changes in real world result in state, in which software system (which was satisfactory in the past) is becoming insufficient. To bring such system to the sufficient state again, it has to be changed. The most common reasons for a change are bug fixes and additions of a new functionality. It is more common to change the existing system than to create a new from scratch. Software system maintenance and evolution consumes up to 80 percent of system's lifetime [16].

To perform a maintenance, it is necessary to understand details related to the requested change correctly. Without sufficient knowledge, it is possible to

break functionality of a system. Therefore a program comprehension is the prerequisite for program maintenance. After understanding the request for a change, the relevant implementation code fragments have to be identified. Just after that these fragments can be safely modified. As a software system is growing, it is more and more difficult to locate code intended for maintenance. This problem raises also with fluctuation of developers.

In this paper, we propose a new architecture for improving a program comprehension, based on utilization of well known classes used in object-oriented program. We claim that for these classes it is possible to define accurate localization algorithm. This algorithm, in combination with a predefined feature knowledge base, maps the selected program feature at a higher level of abstraction automatically. Moreover, this mapping is performed in runtime.

The paper is organized as follows. Section 2 describes the proposed methodology for automatic abstraction of selected feature. Section 3 presents the experiment developed using Java programming language. Section 4 covers the related works. Finally, section 5 concludes the paper and presents future work.

2. Feature Metamodel at Higher Level of Abstraction

Existing tools for dynamic analysis and visualization of system execution are mostly generic solutions. Visualization result is difficult to use, since it is closer to implementation level than to application domain level. To get application domain specific visualization, link between application domain concept and code implementation must exist. This link between different levels of abstraction is not explicitly represented in a program code. When reading and understanding program by a developer, this link is built up using previous developer's experiences and knowledge (about problem domain, programming techniques, algorithms, data structures, etc.). To reproduce this activity automatically, a kind of *knowledge base* describing link between different levels of abstraction must be established.

In this paper, we will focus on how to abstract systems constructed in object-oriented manner. A program developed in object-oriented language is typically defined by group of classes and their instances – objects. Objects communicate to each other by sending messages. Relationships between classes and objects are defined by program code. Let's define V as a set of all existing classes (1) and P as a set of classes used in object-oriented program (2). Figure 1 represents a program created in object-oriented language.

$$V = \{v_1, v_2, \ldots, v_n\} \tag{1}$$

$$P = \{p_1, p_2, \ldots, p_m\}, P \subset V \tag{2}$$

To understand object-oriented program, a developer has to read used classes and understand their relations and meanings. By term understanding we mean *"ability of explaining the program, its structure, its behaviour, its effects on its operational context, and its relationships to application domain in terms that are*

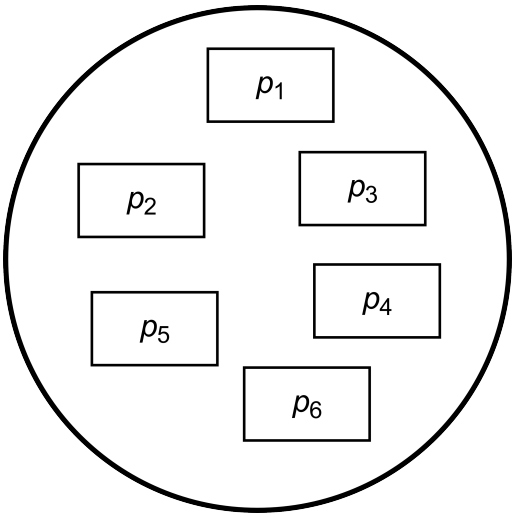

Fig. 1. Object-oriented program using classes p_i

qualitatively different from the tokens used to construct the source code of the program" [2]. It is essential to move implementation level knowledge to a higher level of abstraction. As mentioned above, to make the translation from one level to another, extensive knowledge – a *knowledge base* – is required. Complete knowledge base should contain information about all classes, all application domains and about their relationship. Since usually new classes are defined, it is impossible to prepare complete knowledge base.

However, there is a group of software components, which are well known and their amount is limited – components defined in software libraries. Defining knowledge base which contains information about software libraries, it is possible automatically create the transformation between the implementation level and higher level of abstraction, i.e. such that implements features comprised in libraries.

Modern object oriented software development platforms provides a comprehensive set of its own standard class libraries. Let's designate K as a set of *known classes*, i.e. all classes with known names and meaning. K is a subset of set V, of all classes. (3). Let's define K' as a set of known classes, which are used in the program (4).

$$K = \{k_1, k_2, \ldots, k_r\}, K \subset V \tag{3}$$

$$K' = \{k'_1, k'_2, \ldots, k'_s\}, K' = P \cap K \tag{4}$$

When using classes from standard library, the program becomes easier readable and understandable (Fig. 2).

Let's define *aspect of the program* as a group of known classes used in the program, which relates to one logical part (from higher level abstraction point of

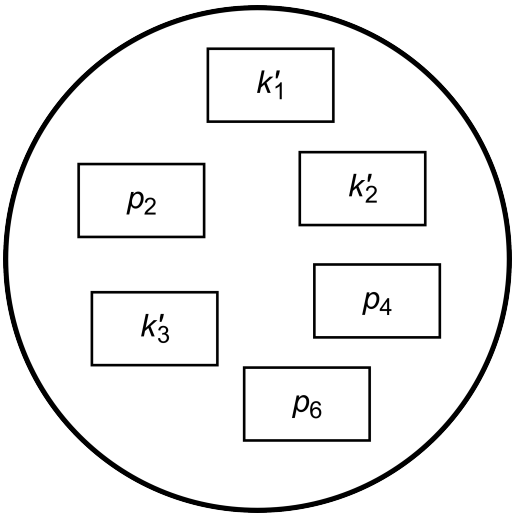

Fig. 2. Object-oriented program using classes p_i and well known classes k_i

view). Aspect of the program is a subset of program's features, since it contains only features based on known classes. Since the program may contain several aspects, A is subset of K' (5). By defining a knowledge base for known classes and by analyzing program for the way of using these classes, it is possible to define a function f, which maps an aspect of the program to model M, which can be created at a higher level of abstraction than implementation level (Fig. 3, equation 6).

$$A = \{a_1, a_2, \ldots, a_t\}, A \subset K' \tag{5}$$

$$M = f(A) \tag{6}$$

In implementation, the knowledge base and the function f will be presented by combination of tracking algorithm and metamodel builder algorithm. Both algorithms will be specific for each feature.

Proposed system architecture is described in fig. 4. Since there are two separate systems (base system and tool), where one react about the other, it is convenient to use a metalevel architecture. Base level is represented by a legacy software system. This system is a subject to analyse. Metalevel represents the tool. Base level system is automatically enhanced with code tracing system runtime execution. Based on tracing results and utilizing knowledge base, the metasystem automatically builds a metamodel of specified feature. This metamodel describes feature implementation at a higher level of abstraction.

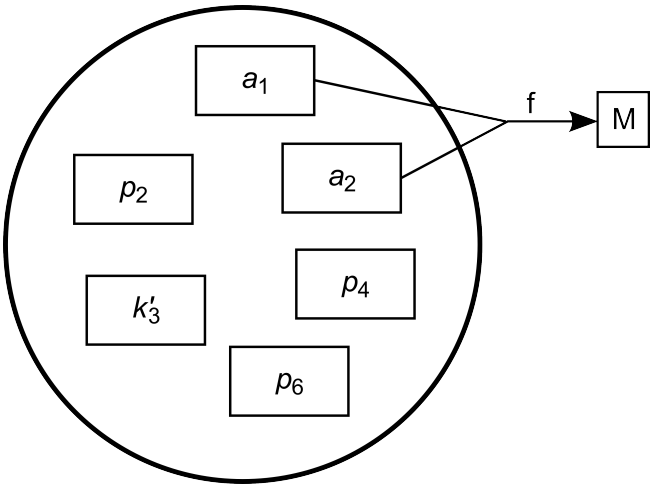

Fig. 3. Object-oriented program using classes p_i and known classes k_i and a_i. By examining the way of use of classes a_i it is possible to create model M at a higher level of abstraction

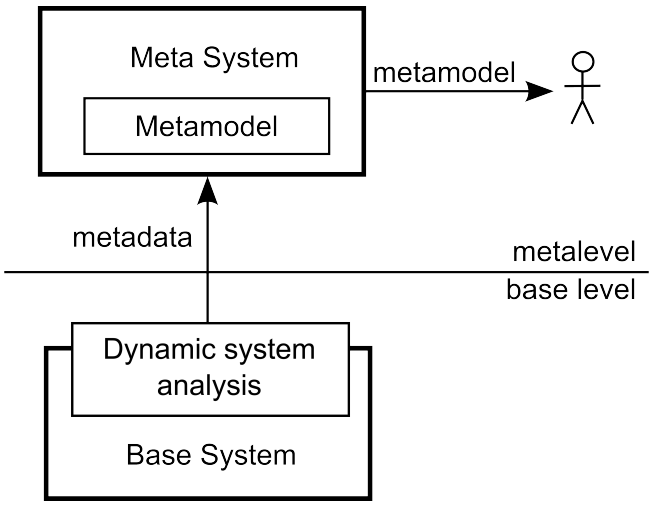

Fig. 4. Metamodel creation using aspect-oriented approach

3. Tool for Feature Metamodel Building

To validate usefulness of our proposal, we have constructed experimental tool, which automatically builds up metamodel for several predefined features. The experiment is performed using Java programming language. This language was chosen because of its wide use and its rich standard library (Java Class Library).

To extend base level application with tracking code, we decided to use aspect-oriented approach. AOP ability for extending existing code with a new functionality is very helpful for techniques depending on metadata – it is possible to extend base system with monitoring code [25]. This new code will create execution traces used to build metamodel.

Process of extending base level application with monitoring code is defined in knowledge base in form of AOP aspects. For each feature, the knowledge base contains information about its implementation – classes and methods used, as well as the way of using these classes and methods. Aspects defined in the knowledge base trace execution and usage of specified classes and methods at the base level of application. According to the traced information, algorithm (also defined in the knowledge base) builds up corresponding metamodel. Aspects and algorithm building metamodel are specific for each modelled feature.

One scenario supported by the experiment is modelling the feature for input/output streams. A stream can be defined as a sequence of data. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays. Basic classes used to work with streams are abstract classes *java.io.InputStream*, *java.io.OutputStream*, that work with bytes, and *java.io.Reader* and *java.io.Writer*, that work on characters. Different extensions of these classes allow different kinds of stream transformations. Among others, class *java.io.ByteArrayInputStream* allows reading bytes from byte array, *java.io.FileInputStream* allows reading bytes from file, *java.io.FilterInputStream* allows stream transformations,
java.util.zip.GZIPInputStream decompresses stream data, etc. In the knowledge base used in experiment, the description of these classes is defined. One recognized way of using these classes is chaining their instances (instance of one class is used as constructor parameter for another one). This is common way for defining the chain of classes used to process the specific stream. For example, stream is read from a file, then it is decrypted and finally decompressed.

Described feature of input/output streams is modelled as the chain of filters used with a stream (Fig. 5).

Fig. 5. Metamodel for input/output streams.

To build such metamodel automatically, all invocations of mentioned classes' constructors must be traced. Metamodel is created according to traced sequence of constructor invocations. Algorithm building metamodel must consider only related invocations (connected through constructor argument). Finally, the tool displays graphical representation of created metamodel (Fig. 6).

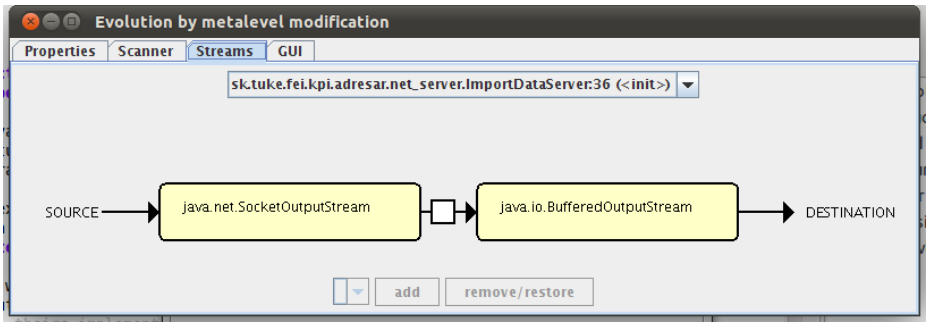

Fig. 6. Data streams metamodel

4. Related Works

The work presented in this paper addresses the issue for improving program comprehension by automatic metamodel abstraction.

According to a new design principle of *open implementation* a software module allows its clients to control its implementation strategy [13].

Open implementation contains besides main interface (providing functionality) also meta-interface through which a client tunes the implementation underlying the primary interface. The wide use of open implementation principle is *metalevel architecture*, see [15].

In general, a metalevel architecture consists of different levels, where one level is controlled by another one. From viewpoint of program represented as a set of objects, it is possible to define several terms in the area of metalevel architectures. Application describing a problem being solved is located at *domain level*. *Domain objects* are objects of this application. These objects describe the problem being solved. *Domain object protocol* defines operations (called *domain operations*) provided by domain object.

Except a domain level there exists a *metalevel*, which provides a space for metaobjects. *Metaobjects* describe, control, implement or modify domain objects. In case of multilevel architecture, metaobject can control another metaobjects. A *metaobject protocol* (MOP) is object-oriented interface allowing communication between objects at domain level and objects at metalevel. It defines application programming interface which can be used to work with metaobjects. Finally, *metaobject operation* is operation from metaobject protocol.

The technology of aspect-oriented programming which allows modularization of *crosscuting concerns* [14], ability for adding a new functionality to an existing program code [25] has been used in our solution as the basic technology for our runtime abstraction.

For maintenance tasks, the first step to be done before a change itself is identifying and understanding program code relevant to the requested change. Identifying parts of source code, that correspond to functional part of program, is known as feature or concept location [32]. *Feature* or *concept* can be defined as cohesive set of functionality [31]. Each feature represents a well understood abstraction of a system's problem domain [28]. It exists at runtime as a collaboration of objects, exchanging messages to achieve a specific goal. The main difference between concepts and features is, that the user can exercise the latter (hence the notion of concept is more general than the notion of feature) [18]. In this work, we focused on features. Features are usually described by the requirements of a software system. Typically, users formulate their requirements, change requests or error reports in terms of features [21].

Finding relations between features and source code takes important part in our program comprehension. Feature location is the process of identifying mappings between domain level concepts and their implementations in source code (it identifies code fragments that implements specific feature) [24]. The maintenance request, expressed usually in natural language and using the domain level terminology is the input of mapping and a set of components that implement the feature [4] is the output. The input and output of the location process belong to different levels of abstraction (domain level on one side, implementation level on the other side). To make the transformation from one level to another, extensive knowledge is required (problem domain, programming techniques, algorithms, data structures, etc.). Since features are not explicitly represented in source code, the features identification is a difficult task. It is traditionally an intuitive and informal process, based on past experiences.

In general, feature location can be static, dynamic or hybrid. The static techniques of feature identification doesn't need execution of subject program – these techniques instead focus on searching in source code. The feature (or concept) location process can be defined as follows [19]:

1. feature formulation (usually in natural language)
2. query formulation and execution based on the intermediary representation
3. investigation of results

First approaches simply utilized pattern matching tools (such as Unix utility grep). This basic principle was improved in several ways ([4] [7] [20] [33] [29] [10] [11]). Improvements include extending the search process with usage of advanced information retrieval methods, or creating different ways of graphical visualisation of query results.

The dynamic technique of feature identification analyzes execution traces of subject program. These techniques are important especially for object-oriented and dynamic languages. With these, it is nearly impossible to obtain a complete

understanding of the system only by inspecting the source code [28]. Object-oriented characteristics such as inheritance and polymorphism make it difficult to understand runtime behavior of the system only by inspecting source code. Finally, hybrid (combined) feature location techniques utilize both - execution trace and source code information. Usually, static information is used to filter the execution traces.

Works [23] [30] [17] use Java Debug Interface to collect execution traces. Using this approach is complicated and leads to performance penalty – so we decided for the use of AOP to create execution traces. In [8] a set of instrumentation aspects is defined that add code to a given Java program to collect enough tracing information such that the program can be reverse engineered. In this approach, every method invocation is intercepted and may be recorded. Since recording each method invocation would yield too much data to be analyzed, the recorder may be customized using a filter expression. However, to define filter expression, implementation level knowledge is required.

Common approach to handle a big amount of execution trace data is its visualization. Standards provided by execution trace analysis are sequence diagrams and mapping method calls to source files. Sequence diagrams, while originally devised as a notation used during analysis and design, can be also very useful in program comprehension – through the visualization of execution call traces. Reverse-engineered sequence diagrams based on dynamic call traces are typically very large, therefore several techniques to handle theirs size were introduced [1].

Works [12] [3] use aspect-oriented programming to instrument executed program with a new code, which collects information about program execution. Collected data are presented as UML sequential diagrams. Paper [22] summarizes experiences with the development of a reverse engineering tool for UML sequence diagrams. Author states that the development of a tool supporting the reconstruction of the behavior of a running software system must address the major areas of data collection, representation of this data in a suitable metamodel, and finally its graphical representation. Work [1] provides an overview of sequence diagram tools.

Paper [9] proposes a reverse-engineering tool suite to build precise class diagrams from Java programs. The tool uses both static and dynamic data to infer relationships among classes and interfaces. Work [3] describes class diagrams used to create metamodels. These metamodels holds collected data and model resulting diagrams.

Besides UML diagrams, several tools uses own specific way of data visualization. Paper [5] proposes trace visualization techniques based on the massive sequence and circular bundle view. Paper [6] introduces a novel 3D visualization technique that supports animation of feature behavior. The approach exploits a third dimension to visually represent the dynamic information, namely object instantiations and message sends. Work [26] describes user views of the execution that are specific to the program being understood and to the particular problem.

Described ways of dynamic analysis and visualization of system execution are generic solutions. Such approaches are often difficult to use, since it is impossible to have only one simple general visualization suitable for different kinds of specific behaviours. Therefore to use such tools, user must understand (often complex) visualization output, which is closer to implementation level than to application domain level. As stated in [26] and [27], understanding the software behavior is a unique problem requiring a specialized solution and a visualization.

Our approach comes out from the fact introduced above. One of our aims was to provide feature specific visualization, which will be closer to application domain level than to implementation level. To create such a visualization, feature specific model is built up during dynamic analysis of system execution.

5. Conclusions and Future Work

The paper presents a new approach to automatic feature mapping between different levels of abstraction. The proposed method utilizes knowledge of standard class libraries. The experimental tool based on this method has been developed, to prove the ability for automatic feature visualization at a higher level of abstraction than that of the implementation. Based on the usage of known classes, the tool is able to automatically track down selected predefined feature of existing application and create its metamodel. Since the metamodel is specific for tracked feature, also its visualization can be closer to application domain and better describes given problem (in contrary to generic visualizations). Predefined knowledge base substitutes developer's knowledge.

The development of such automatic tool is not so simple. The main difficulty is associated with knowledge base construction – for each recognized feature there must be aspects defined to trace feature implementation and algorithms to model traced implementation details in metamodel. The level of difficulty of aspects and algorithms depends on the specific feature. On the other side, once the tool is developed, it can be used to locate predefined features from any existing application using compatible version of libraries. It is even possible to use the tool with application with no source code available (in case of using load-time weaving of aspects). The prerequisite is just the compatibility of application programming interface.

Acknowledgments. This work was supported by VEGA Grant No. 1/0015/10 Principles and methods of semantic enrichment and adaptation of knowledge-based languages for automatic software development.

References

1. Bennett, C., Myers, D., Storey, M.A., German, D.M., Ouellet, D., Salois, M., Charland, P.: A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. J. Softw. Maint. Evol. 20, 291–315 (July 2008), http://portal.acm.org/citation.cfm?id=1400155.1400156

2. Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: Program understanding and the concept assignment problem. Commun. ACM 37, 72–82 (May 1994), http://doi.acm.org/10.1145/175290.175300
3. Briand, L.C., Labiche, Y., Leduc, J.: Toward the reverse engineering of uml sequence diagrams for distributed java software. IEEE Trans. Softw. Eng. 32, 642–663 (September 2006), http://portal.acm.org/citation.cfm?id=1248725.1248762
4. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: Proceedings of the 8th International Workshop on Program Comprehension. pp. 241–. IWPC '00, IEEE Computer Society, Washington, DC, USA (2000), http://portal.acm.org/citation.cfm?id=518049.856972
5. Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., van Wijk, J.J.: Execution trace analysis through massive sequence and circular bundle views. J. Syst. Softw. 81, 2252–2268 (December 2008), http://portal.acm.org/citation.cfm?id=1454787.1454981
6. Greevy, O., Lanza, M., Wysseier, C.: Visualizing live software systems in 3d. In: Proceedings of the 2006 ACM symposium on Software visualization. pp. 47–56. SoftVis '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1148493.1148501
7. Griswold, W.G., Yuan, J.J., Kato, Y.: Exploiting the map metaphor in a tool for software evolution. In: Proceedings of the 23rd International Conference on Software Engineering. pp. 265–274. ICSE '01, IEEE Computer Society, Washington, DC, USA (2001), http://portal.acm.org/citation.cfm?id=381473.381501
8. Gschwind, T., Oberleitner, J.: Improving dynamic data analysis with aspect-oriented programming. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering. pp. 259–. IEEE Computer Society, Washington, DC, USA (2003), http://portal.acm.org/citation.cfm?id=872754.873577
9. Guéhéneuc, Y.G.: A reverse engineering tool for precise class diagrams. In: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. pp. 28–41. CASCON '04, IBM Press (2004), http://portal.acm.org/citation.cfm?id=1034914.1034917
10. Hill, E.: Developing natural language-based program analyses and tools to expedite software maintenance. In: Companion of the 30th international conference on Software engineering. pp. 1015–1018. ICSE Companion '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1370175.1370226
11. Hill, E., Pollock, L., Vijay-Shanker, K.: Automatically capturing source code context of nl-queries for software maintenance and reuse. In: Proceedings of the 31st International Conference on Software Engineering. pp. 232–242. ICSE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICSE.2009.5070524
12. Khaled, R., Noble, J., Biddle, R.: Inspectj: program monitoring for visualisation using aspectj. In: Proceedings of the 26th Australasian computer science conference - Volume 16. pp. 359–368. ACSC '03, Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2003), http://portal.acm.org/citation.cfm?id=783106.783147
13. Kiczales, G.: Beyond the black box: Open implementation. IEEE Softw. 13, 8–11 (January 1996), http://portal.acm.org/citation.cfm?id=624611.625543
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. pp. 220–242 (1997)
15. Kočí, R.: Methods and Tools for Implementation of Open Simulation Systems. Ph.D. thesis, Brno University of Technology, Brno, Czech Republic (2004), http://www.fit.vutbr.cz/research/view_pub.php?id=7613

16. Lehman, M.M., Ramil, J.F., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Technical report 2001/8, Imperial College, Department of Computing, London, United Kingdom (September 2001)
17. Leroux, H., Réquilé-Romanczuk, A., Mingins, C.: Jacot: a tool to dynamically visualise the execution of concurrent java programs. In: Proceedings of the 2nd international conference on Principles and practice of programming in Java. pp. 201–206. PPPJ '03, Computer Science Press, Inc., New York, NY, USA (2003), http://portal.acm.org/citation.cfm?id=957289.957349
18. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 234–243. ASE '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1321631.1321667
19. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A.: Static techniques for concept location in object-oriented code. In: Proceedings of the 13th International Workshop on Program Comprehension. pp. 33–42. IEEE Computer Society, Washington, DC, USA (2005), http://portal.acm.org/citation.cfm?id=1058432.1059343
20. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering. pp. 214–223. IEEE Computer Society, Washington, DC, USA (2004), http://portal.acm.org/citation.cfm?id=1038267.1039053
21. Mehta, A., Heineman, G.T.: Evolving legacy systems features using regression test cases and components. In: Proceedings of the 4th International Workshop on Principles of Software Evolution. pp. 190–193. IWPSE '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/602461.602507
22. Merdes, M., Dorsch, D.: Experiences with the development of a reverse engineering tool for uml sequence diagrams: a case study in modern java development. In: Proceedings of the 4th international symposium on Principles and practice of programming in Java. pp. 125–134. PPPJ '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1168054.1168072
23. Oechsle, R., Schmitt, T.: Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). In: Revised Lectures on Software Visualization, International Seminar. pp. 176–190. Springer-Verlag, London, UK (2002), http://portal.acm.org/citation.cfm?id=647382.724668
24. Olszak, A., Jørgensen, B.N.: Remodularizing java programs for comprehension of features. In: Proceedings of the First International Workshop on Feature-Oriented Software Development. pp. 19–26. FOSD '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1629716.1629722
25. Oriol, M., Cazzola, W., Chiba, S., Saake, G.: Object-oriented technology. ecoop 2008 workshop reader. chap. Getting Farther on Software Evolution via AOP and Reflection, pp. 63–69. RAM-SE'08, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02047-6_7
26. Reiss, S.P.: Visualizing program execution using user abstractions. In: Proceedings of the 2006 ACM symposium on Software visualization. pp. 125–134. SoftVis '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1148493.1148512
27. Reiss, S.P.: Visual representations of executing programs. J. Vis. Lang. Comput. 18, 126–148 (April 2007), http://portal.acm.org/citation.cfm?id=1230158.1230422
28. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Feature driven browsing. In: Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007. pp. 79–100. ICDL '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1352678.1352684

29. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th international conference on Aspect-oriented software development. pp. 212–224. AOSD '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1218563.1218587
30. Sundararaman, J., Back, G.: Hdpv: interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In: Proceedings of the 4th ACM symposium on Software visualization. pp. 47–56. SoftVis '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1409720.1409729
31. Turner, C.R., Fuggetta, A., Lavazza, L., Wolf, A.L.: A conceptual basis for feature engineering. J. Syst. Softw. 49, 3–15 (December 1999), http://portal.acm.org/citation.cfm?id=340287.351492
32. Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. Journal of Software Maintenance 7, 49–62 (January 1995), http://portal.acm.org/citation.cfm?id=249936.249939
33. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: Sniafl: Towards a static noninteractive approach to feature location. ACM Trans. Softw. Eng. Methodol. 15, 195–226 (April 2006), http://doi.acm.org/10.1145/1131421.1131424

Michal Vagač is Assistant Professor at Department of Informatics, Faculty of Natural Sciences, Matej Bel University, and PhD student at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science, in 2001. The subject of his research is metamodeling, metaprogramming, programming paradigms, and dynamic software systems adaptation.

Ján Kollár is Full Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his M.Sc. summa cum laude in 1978 and his Ph.D. in Computer Science in 1991. In 1978-1981 he was with the Institute of Electrical Machines in Košice. In 1982-1991 he was with Institute of Computer Science at the P.J. Šafárik University in Košice. Since 1992 he is with the Department of Computer and Informatics at the Technical University of Košice. In 1985 he spent 3 months in the Joint Institute of Nuclear Research in Dubna, USSR. In 1990 he spent 2 months at the Department of Computer Science at Reading University, UK. He was involved in research projects dealing with real-time systems, the design of microprogramming languages, image processing and remote sensing, dataflow systems, implementation of programming languages, and high performance computing. He is the author of process functional programming paradigm. Currently his research area covers formal languages and automata, programming paradigms, implementation of programming languages, functional programming, and adaptive software and language evolution.

Received: April 6, 2011; Accepted: August 17, 2011.