EVOLVING METAMODELS IN ASPECT-ORIENTED MANNER

Michal VAGAČ*, Ján KOLLÁR**, Sergej CHODAREV**
*Department of Informatics, Faculty of Natural Sciences, Matej Bel University,
Tajovského 40, 974 01 Banská Bystrica, Slovak Republic, e-mail: michal.vagac@gmail.com
**Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic, e-mail: jan.kollar@tuke.sk

ABSTRACT

Multi-layered software architecture allows to allocate different responsibilities to different layers. When such a responsibility is to react about other part of the system, we can speak about a metalevel architecture. Such an architecture consists of at least two levels–a base level and a metalevel, where the metalevel reasons about the base level. A causal connection associates the base level objects with the metalevel objects and guarantees that changes to the metalevel are reflected into corresponding changes to the base level and vice-versa [14].

This paper presents an innovative approach to handle casual connections in metalevel architectures. The base level is represented by a legacy application. The metalevel contains a metamodel. The metamodel represents certain feature of the base level application. Aspect-oriented techniques are used to add a new code to the base level application. This code manages casual connection between the base level and the metamodel at the metalevel.

Keywords: *aspect-oriented programming, metalevel architecture, metaprogramming, software change*

1. INTRODUCTION

In a multi-layered software architecture, different responsibilities of an application are allocated to different layers. This is useful especially in more complex applications, where it makes maintenance easier. When one layer is subject of another layer, we can speak about metalevel architecture. Then metalevel controls, handles, or describes the base level.

In general, a metalevel architecture consists of different levels, where one level is controlled by another one (Fig. 1). From the view of object-oriented programming, where a program is represented as a set of objects, it is possible to define several terms in area of metalevel architectures. Application describing problem being solved is located at *domain level*. *Domain objects* are objects of this application. These objects describe the problem being solved. A *domain object protocol* defines operations provided by a domain object. A *domain operation* is an operation from a domain object protocol.

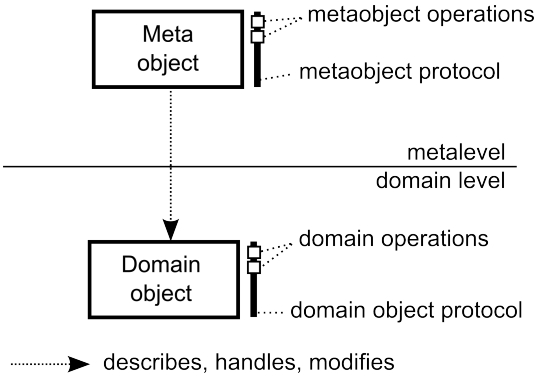

Fig. 1 Metalevel architecture from view of object-oriented programming

Besides the domain level there is a *metalevel* which provides space for metaobjects. *Metaobjects* describe, control, implement or modify domain objects. In the case of a multilevel architecture, a metaobject can control other metaobjects. A *metaobject protocol* (MOP) is object-oriented interface allowing communication between objects at the domain level and objects at the metalevel. It defines application programming interface which can be used to work with metaobjects. Metaobject protocols in a programming language are interfaces to the language which provide ability to change language behavior and language implementation [9]. Finally, a *metaobject operation* is an operation from the metaobject protocol.

The metalevel contains internal structure (*metadata*) which describes the base level. The metalevel is *casually connected* to its base level, when any change in the base level reflects in the corresponding change of metadata and vice-versa–any change in metadata will be reflected at the base level.

The term meta in general expresses information about information. Metaprogramming relates to programs which manipulate other programs. Metalevel contains data which are representing related part of base level. If this representation always corresponds to real state of the base level, we can say that the base level and the metalevel are casually connected.

Aspect-oriented programming allowed better separation of such concerns which are impossible to describe by available language constructs (as classes or methods) used to modularize code. These concerns, named *crosscutting concerns*, are impossible to modularize from its essence, since they crosscut basic system functionality. This results in tangled code which it is difficult to understand and reuse. Two concerns crosscut when they have to be composed differently, but at the same time they must be coordinated [10].

An *aspect* represents a modular unit consisting of the *pointcut* and the *advice*. *Join points* are points in which an aspect crosscuts a basic program. By defining a *pointcut*, it is possible to define a set of join points. *Advice* allows to define an action executed at points defined by the point-

cut. Composing defined aspects and the affected program is a task of the aspect weaver. The most common way of aspect implementation is weaving the aspect code into the program code.

Besides clearer modularization possibilities, aspect-oriented programming allowed adding a new functionality to an existing code.

This paper describes innovative approach which uses aspect-oriented techniques to handle casual connection between the base level and the metalevel. A base level application is advised with a new code which gathers runtime information about the application, and after change request it introduces a new code which extends (or replaces) the original one.

The rest of the paper is organized as follows: Section 2 describes our method of casual connection utilizing aspect-oriented techniques; Section 3 describes an experimental tool created according to the method proposal. Section 4 provides a brief overview of topics related to our work. Finally, Section 5 draws our conclusions resulting from the text presented.

2. CASUAL CONNECTION USING AOP

From the user's point of view, an application is represented by a set of functional parts–*features*. Each feature represents a well-understood abstraction of a system's problem domain [18]. It exists at runtime as a collaboration of objects exchanging messages to achieve a specific goal.

The method proposed uses a multilevel architecture to model relation between application feature implementation and its model. The base level is represented by a legacy application. The metalevel contains different types of metamodels of selected features from the base level. When handling casual connection between the base level and the metalevel (feature and its implementation), two situations have to be handled: information transfer from the base level to the metalevel and vice-versa–information transfer from the metalevel to the base level.

When handling *information transfer from the base level to the metalevel*, it is required to get all information about feature implementation in the base level application. Since this information must be gathered during runtime, it is impossible to use any static code analysis techniques. Instead, aspect-oriented programming ability of extending existing program with a new code is used. By using aspect-oriented programming, the base level application is extended with a code which monitors execution of the base level application. The new code is added to those parts of the application which relates to the feature implementation. While running the base level application, monitoring code gathers all needed information and sends it to the metalevel. At the metalevel, according to this information, the metamodel is created or updated.

Handling *information transfer from the metalevel to the base level* is more difficult. After metamodel change, it is required to alter the base level application. Changing in general a running program is a difficult task. Among other things, the biggest issues to solve are handling of active threads and transfer of a program state. Replacing existing class with another one can break the functionality of the original program (e.g. when removing methods which are used by other classes). The method proposed uses aspect-oriented techniques also for application of change. With help of adding a new code to an existing application, it is possible to use around advice to avoid execution of selected parts of original code. Instead of original code, it is possible to get executed a new code implementing change (Fig. 2). This way it is possible to apply required changes without need of solving problems related to general dynamic software change.

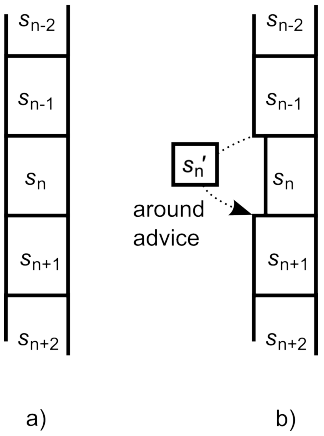

Fig. 2 Original sequence of statements a) is advised with a new code b). This code avoids statement S_n and instead uses the new statement S_n'

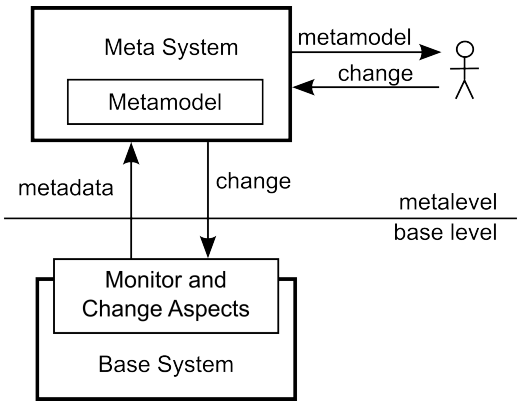

Fig. 3 Overall system architecture–relation between the base level and the metalevel

Fig. 3 describes the overall system architecture. The base level application is extended with aspects which monitor its execution and change its behavior. While executing the application, the monitor aspect is gathering data about execution. These data are sent to the metalevel. There the data are used to create (or update if already exists) a metamodel. As follows, the metamodel is presented to the user.

After changing the metamodel, the change aspects are used to propagate the change to the base level application. Affected code is advised with around aspect. According to the type of change, the original code is completely avoided or extended with the new functionality.

ISSN 1335-8243 (print) © 2011 FEI TUKE
www.aei.tuke.sk

ISSN 1338-3957 (online)
www.versita.com/aei

3. METAMODEL OF DATA STREAMS

The Experiment based on the method described consists of two parts–a base level application and a completely independent metasystem. Both levels are developed using Java programming language.

The base level application focuses on work with data streams. Input-output data stream is a sequence of data. Input stream reads data from a defined source (file, network, memory, etc.) and transfers it to the program; output stream gets data from the program and transfers it to the defined target (file, network, memory, etc.).

Main Java classes dedicated to work with data streams are abstract classes *java.io.InputStream*, *java.io.OutputStream*, *java.io.Reader* and *java.io.Writer*. The first two classes work with bytes, the second ones work with characters (thus support encoding). In the experiment, we have focused only on byte-related classes. Java API defines several classes which extend mentioned abstract classes and which allows processing of data which passes via the stream.

It is possible to make combinations of classes mentioned. The first class can be connected with the second one by passing itself as a constructor parameter while creating the second class. This way it is possible to create a "chain" of classes through which the stream of data passes. Each "link" in the chain has the ability to process passed data in some way (for example to do any compression and encryption).

In the experiment, the task of the metalevel is to create metamodels of data stream chains used in the program. Each metamodel represents chain of processors affecting data passed in the stream (Fig. 4).

Fig. 4 Data stream metamodel

The base level application has no information about the metalevel–and so there are no extra requirements on its development. It simply implements tasks related to its domain. The metalevel uses abilities of aspect-oriented programming (namely AspectJ) to create a relation to the base level. As stated above, the solution of casual connection between two levels is handled in two parts: gathering information about the base level and changing the base level application behavior.

To get information about the actual data streams implementation, it was necessary to catch all information about all used classes which inherit from the one of *java.io.InputStream* or *java.io.OutputStream*. It was done by pointcuts defined for all constructors for the mentioned types of classes. Always when a new instance of such a class is created, defined advice checks for existence in the related metamodel. If the tracked class is not defined yet in the particular metamodel, a new "chain link" in metamodel is created. If the class is already defined, it is verified against the existing metamodel. The essential part is context tracking of intercepted constructor invocations which defines relationship between created instances. This way a metamodel is automatically created, while running the base level application.

To change the behavior of the base level program is a more difficult task. It is possible to realize two types of changes–to insert a new instance of a class processing a data stream, and to remove (or disable) an existing instance of a class processing a data stream. This problem partly is solved by applying the Cuckoo's Egg aspect-oriented design pattern [16]. The pattern defines an aspect that intercepts the creation of the original class and instead returns an instance of replacement class transparently to the original business logic.

Described aspect is defined for each creation of an instance of the stream processing class. Its around advice allows to make the execution of original code conditional. When there is a request to remove the "chain link", the around advice simply avoid using of the disabled class. In the case of the other option–insertion of a new instance–this instance is added before (or after) execution of the advised original class constructor.

The result of the experiment described is the existence of a tool which allows to create a metamodel of data streams used in any Java application. The application doesn't need to be prepared or changed in any way. The tool is linked with the application by aspect-oriented techniques. When running the application, the tool automatically creates metamodels of used data streams. These metamodels can be changed (existing stream processors can be removed or new added) and this change is automatically reflected in the base level application behavior.

4. RELATED WORK

Tracing a base level application to collect information about it is not a new idea. One possibility to trace an application in Java language is using Java Debug Interface (JDI). With a help of this approach the application doesn't need to be modified. The collected data are usually analysed and/or visualised. This approach is used in several works [13, 17, 19].

Another possibility to collect information about a running system is using the mentioned properties of aspect-oriented programming. [6, 11] uses AOP to instrument subject system with a new tracing code. The collected data are visualised to a user. The most common way to visualise such data are UML sequential diagrams. [6] describes class diagrams used to store metamodels built from the collected data. Papers [8, 12, 15] contain overview of other approaches used to reconstruct a software system behavior.

The mentioned works create different kinds of metamodels of the collected data. The metamodel modifications are not supported.

An idea of a program modification utilizing AOP was used in [7]. Authors of the paper are using aspects to incorporate new, collaborative features into existing applications. Modification of existing applications with help of AOP is subject also of several other works [2–5]. In the first step, a subject application is extended with a tracing code. As follows the application is executed. After navigating the ap-

plication flow to the point of interest, the tracing code is enabled. Gathered trace information is filtered and analysed–the result of the analysis are the class and the method which will be modified. Modification itself is described by an aspect, which is defined according to the analysis result and according to the required change.

Works [1,20] differentiate two types of changes–domain specific change types and generally applicable change types. A domain specific change is described in a domain specific way. A generally applicable change can be a kind of an aspect-oriented design pattern. The relationship between these two groups is maintained in a catalog of changes. Each domain specific change type is defined as a specialization of one or more generally applicable changes.

5. DISCUSSION/CONCLUSIONS

Aspect-oriented programming allowed adding a new functionality to an existing code. The approach presented uses this aspect-oriented programming property to handle casual connections in a metalevel architecture. The base level of this architecture is automatically extended with two types of advices. The first one automatically tracks down all information about the monitored feature implementation. According to this information, metamodel representing the feature is created at the metalevel. After the metamodel change, the second type of an advice is used to reflect changes in the base level application. The change is mostly implemented by AOP around advice which allows to make execution of the original code conditional. When needed, the original code is only extended, when needed, it can be completely avoided.

The experiment described confirmed the method proposed. The tool is able to automatically create a metamodel of the specified program feature. After changing the metamodel, the program behavior is changed. As a result the casual connection is accomplished–the metamodel always reflects state of the base level and vice-versa–changes in the metamodel are always reflected in the base level.

ACKNOWLEDGEMENT

This work is the result of the project implementation: Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120030) supported by the Research & Development Operational Program funded by the ERDF.

REFERENCES

[1] BEBJAK, M. – VRANIĆ, V. – DOLOG, P.: *Evolution of web applications with aspect-oriented design patterns*. In M. Brambilla and E. Mendes, editors, Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007, Como, Italy, July 2007, pp. 80–86.

[2] BLUEMKE, I. – BILLEWICZ, K.: *Aspect oriented programming in program tracing and modification*. Agility and discipline in software engineering, Nakom, Poznan, 2007, pp. 23–34.

[3] BLUEMKE, I. – BILLEWICZ, K.: *Aspects modification in business logic of compiled Java programs*. IEEE First International Conference on Information Technologies, Gdansk, Poland, May 2008, pp. 409–412.

[4] BLUEMKE, I. – BILLEWICZ, K.: *Aspects in the Maintenance of Compiled Programs*. IEEE 3rd International Conference on Dependability of Computer Systems DepCoS 2008, pp. 253–260.

[5] BLUEMKE, I. – BILLEWICZ, K.: *Aspect Modification of an EAR Application*. CIS2E 08, Krakow, Poland, 2008, Springer.

[6] BRIAND, L. C. – LABICHE, Y. – LEDUC, J.: *Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software*. IEEE Trans. Softw. Eng., September 2006, 32:642–663.

[7] CHENG, L.-T. – PATTERSON, J. – ROHALL, S. L. – HUPFER, S. – ROSS, S.: *Weaving a Social Fabric into Existing Software*. In Proceedings of the 5th International conference on Aspect-oriented software development AOSD05, March, Chicago, USA, 2005, pp. 147–159.

[8] HAMOU-LHADJ, A. – LETHBRIDGE, T. C.: *A survey of trace exploration tools and techniques*. In Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, CASCON 04, IBM Press, 2004, pp. 42–55.

[9] KICZALES, G. – RIVIERES, J. D. – BOBROW, D. G.: *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, July 1991.

[10] KICZALES, G. – LAMPING, J. – MENDHEKAR, A. – MAEDA, Ch. – LOPES, C. V. – LOINGTIER, J.-M. – IRWIN, J.: *Aspect-oriented programming*. In *ECOOP*, 1997, pp. 220–242.

[11] KHALED, R. – NOBLE, J. – BIDDLE, R.: *InspectJ: program monitoring for visualisation using aspectJ*. In Proceedings of the 26th Australasian computer science conference - Volume 16, ACSC 03, Darlinghurst, Australia, Australia, 2003, Australian Computer Society, Inc., pp. 359–368.

[12] KOLLMAN, R. – SELONEN, P. – STROULIA, E. – SYSTÄ, T. – ZUNDORF, A.: *A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering*. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE02), Washington, DC, USA, 2002, IEEE Computer Society.

[13] LEROUX, H. – RÉQUILÉ-ROMANCZUK, A. – MINGINS, Ch.: *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, PPPJ 03, New York, NY, USA, 2003, Computer Science Press, Inc., pp. 201–206.

[14] MEERSMAN, R.: *On the move to meaningful internet systems*. CoopIS, DOA, and ODBASE : OTM

Confederated International Conferences, Agia Napa, Cyprus, October 31-November 4, 2005, ISBN 978-3-540-29738-3.

[15] 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, PPPJ 06, New York, NY, USA, 2006, ACM, pp. 125–134.

[16] MILES, R.: *AspectJ Cookbook*. December 2004, O'Reilly Media, ISBN 978-0-596-00654-9.

[17] 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, London, UK, 2002, Springer-Verlag, pp. 176–190.

[18] 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, ICDL '07, New York, NY, USA, 2007, ACM, pp. 79–100.

[19] 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, SoftVis 08, New York, NY, USA, 2008, ACM, pp. 47–56.

[20] VRANIĆ, V. – MENKYNA, R. – BEBJAK, M. – DOLOG, P.: *Aspect-Oriented Change Realizations and Their Interaction*. e-Informatica Software Engineering Journal, 3(1):43–58, 2009.

Received June 16, 2011, accepted September 16, 2011

BIOGRAPHIES

Michal Vagač is Assistant Professor at the 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 the 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 the 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.

Sergej Chodarev is PhD student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Computer Science in 2009. The subject of his research is domain-specific languages, metaprogramming and programming paradigms.