A Dynamic Software Evolution by Metamodel Change

Michal Vagač*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
michal.vagac@gmail.com

Abstract

Every long-time running software system is sooner or later subject of a change. The most common reasons are different requests for a bug fixing or adding a new functionality. Software maintenance forms bigger part of software's lifetime. Before applying a change, it is essential to correctly understand current state of affected system. Without all relevant information about both – system as whole and implementation details, a change can introduce new bugs or even break functionality of the system. In the paper we present contribution to program comprehension and following program change. Our method utilizes metalevel architectures to separate legacy application from evolution tool. The tool, placed in metalevel, uses aspect-oriented techniques to introduce a new code in the base level legacy application. This code manages casual connection between base level and metamodel, which is automatically created in metalevel. According to the base level program behavior, the metamodel is created and/or updated. Depending on the metamodel change, the base level program is extended with code which affects its functionality. Since the metamodel describes related implementation in higher level of abstraction, the method improves program comprehension and simplifies change.

Categories and Subject Descriptors

D.1.5 [**Software Engineering**]: Object-oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

*Recommended by thesis supervisor: Prof. Ján Kollár. Defended at Faculty of Electrical Engineering and Informatics, Technical University of Košice on September 29, 2011.

Vagač, M. A Dynamic Software Evolution by Metamodel Change. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 3, No. 3 (2011) 36-43

Keywords

Aspect-oriented programming, metalevel architecture, metaprograming, program comprehension, software evolution, software change

1. Introduction

Information technologies influence many aspects of our lives. Many fields of society rely on computers and software they are running. None software is perfect and sooner or later there are requirements for its change. The most common reasons are different kinds of bug fixing or adding a new functionality. Very often software system models real world problem; since real world also changes, there is a demand to reflect these changes in corresponding software system. It is more common to change existing system than to create a new one. Software maintenance forms about 60-90% of its lifetime [5, 4].

According to mentioned facts, a software maintenance become important research field. All improvements in software change influences time required to maintenance (and therefore also overall cost). Before applying a change, it is required to identify code affected by the change. It is getting more difficult as software systems continue to grow. The problem is even more serious because of fluctuation of developers. Program comprehension is a prerequisite for program maintenance.

While reading and understanding a program code, a developer gains new knowledge. This knowledge can be described as a sequence of certain knowledge domains, where solved problem is on one side, and program implementation solving the problem is on the other side [2]. Knowledge domains on one side are human related, on the other side are computer related. Thus the way of expression is different – on one side it is freely formed, while on the other side it is strictly qualified. During program comprehension, it is required to find associations between both these ends. Relation between two neighbor domains must not be necessarily 1:1 – for example an operation in an algorithm domain can be described by sequence of statements in programming language domain. Different domains describe solved problem in different levels of abstraction. A person understands a program, when he/she is able to explain the program and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program [1].

In the most of cases, applying a change means identifying relevant fragment of source code, changing source

code, shutting down running system and finally replacing old version with a new one. Better approach would be to have a general technique monitoring running system. According to collected information presented to a user, it would be possible to apply modification without stopping the system. A new version of the system would be again monitored (and later again modified) using the same general technique.

In the paper we present a contribution in the field of program comprehension and evolution. The major goal was to find a way to allow automatic mapping between different levels of abstraction of selected program feature. This can be described by two main goals:

- Mapping information from implementation level to a metamodel created in higher level of abstraction.
- Reflecting metamodel changes as modifications in implementation level.

These two levels – implementation level and metamodel in metalevel – must be casually connected. Metamodel in metalevel must reflect actual state of implementation level and vice-versa – all changes in metamodel must be automatically reflected in implementation level.

2. Improving program change

Applying program change requires far more activities than program code modification. Usually a change request is described in terms related to problem domain. Before code modifications, an actor must understand problem domain (to qualify change impact on other parts in problem domain). Then relations between problem domain and program code must be found. To be able to find these relations, an actor must be familiar with used programming languages. To understand how the change will affect running system, also runtime environment must be understood. As a result from stated, a program change heavily depends on knowledge in different areas – programming techniques on one side and problem domain on the other. The most of these prerequirements depends on the actor of change – mostly on his/hers knowledge and experiences. To improve possibilities of program change, it is needed to move some of these responsibilities from a user (developer) to a computer.

Process of identifying fragments of program code related to known functionality of program is known as *feature or concept location* [11]. Feature (or concept) can be defined as cohesive set of functionality of the system [10]. Each feature represents a well understood abstraction of a system's problem domain [9]. This term is situated in higher level of abstraction than source code. During program execution it exists as a collaboration of objects, which are 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) [6]. Later in this paper we will use only term "concept".

Finding relations between concepts and program code takes important part in overall program comprehension. The task is to identify mappings between domain level concepts and their implementations in source code [7]. The input of the mapping is the maintenance request, expressed usually in natural language and using the domain level terminology. The output of the mapping is a set of components that implement the concept [3]. The input and output of location process belong to different levels of abstraction (domain level vs implementation level). To make the translation from one level to another possible, extensive knowledge is required (problem domain, programming techniques, algorithms, data structures, etc.). Since concepts are not explicitly represented in source code, the concepts identification is a difficult task. Concept location directly supports software maintenance.

All proposed solutions of concept location are at most semi-automatic. The main reasons are missing link between different levels of abstraction (concepts are not explicitly represented in a program code) and demand on extensive knowledge base (which is provided by human user). Semi-automated tool ability of searching and analyzing subject system code is combined with user's knowledge (let's name this knowledge as *knowledge base*).

In this work, we will focus on object-oriented programming. 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 set of all existing classes (1) and P as a set of classes used in a program (2).

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

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

To understand an object-oriented program, a developer has to read used classes and understand their relations and meanings. It is essential to move implementation level knowledge to higher level of abstraction. As mentioned above, to make the translation from one level to another, extensive knowledge – a knowledge base – is required. If the program is composed of classes present in the knowledge base, a user can read and understand the program. If there are unknown classes (not present in the knowledge base), a user have to study these classes (and complete his/her knowledge base). According to all these information (usage of classes and their meanings), it is possible to describe a problem in terms in higher level of abstraction than is implementation level.

Automatic creation of concept representation in higher level of abstraction than concept implementation requires both – information about way of using different classes, and also meaning of these classes. A program comprehension with help of semi-automated tool for concept location is accomplished in following steps:

1. Defining search query or execution test case (both related to located concept and utilizing developer's knowledge base).
2. Using the query to search source code or executing the test case.

3. According to developer's knowledge base, deciding, if the results are satisfactory or not. If not, repeating from first step; if yes, getting a mapping from concept to its implementation (mapping between different levels of abstractions).

As can be seen from previous steps, the knowledge base is essential part of program comprehension. This part is also the most problematic (from automation point of view). It is too general to be completely provided by a computer. Also there is always possibility to define new classes – so knowledge base cannot contain all possible information.

However, creating software systems is rarely solely defining new classes. Significant part of many software systems consists of using existing libraries, or reusing other kind of code (maybe created in previous projects). Therefore, defining a knowledge base may help with software comprehension and later modification. The knowledge base for at least limited group of known classes could allow to automatically create higher level of abstraction of usage of these classes.

Let's define K as a set of known classes (with known name and known meaning). K is a subset of V (3). Let's define K' as a set of known classes, which are used in the program (4).

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

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

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 view). Aspect of the program is subset of program's concepts, since it contains only concepts based on known classes. Aspect of the system is a logical part of the system (for example network communication, file system operations, etc) which is defined by known classes. Since the program may contain several aspects, A is subset of K' (5).

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

By defining a knowledge base for known classes and by analyzing program for the way of using these classes, let's define a function f, which projects aspect of the program on model M, which is in higher level of abstraction than implementation level (Fig. 1, equation 6).

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

Same way it is possible to define function g, which applies all model M changes to classes from set A (Fig. 2).

3. System Evolution by Metamodel Change

In our method, one part of the system will reason about the other part. We decided to use properties of metalevel architectures, where one layer is subject of another layer. A metalevel controls, handles, or describes

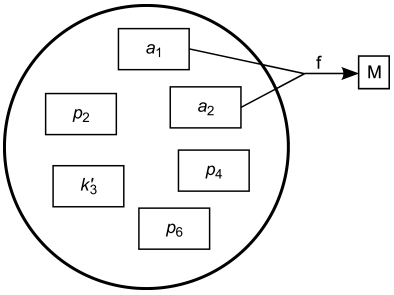

Figure 1: Object-oriented program using classes p_i and known classes k_i and a_i. By examining the way of use of classes a_i it is possible to create model M in higher level of abstraction.

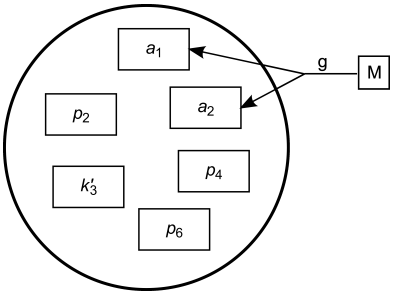

Figure 2: Object-oriented program using classes p_i and known classes k_i and a_i. Changes in metamodel M are reflected in implementation of classes a_i.

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

3.1 Automatic Metamodel Creation

In our method, the base level system of metalevel architecture will be represented by a legacy application. Information gathered by monitoring this base level system will be used to create a metamodel, which will be placed in metalevel (Fig. 3). A *Metamodel* presents identified concept of software system. Since metamodel usually describes more than one object in base level (usually it will contain several objects and relations between them), we decided to define a new term of metamodel instead of using a term metaobject. As follows, it is possible to define terms *metamodel operations* – operations supported by metamodel, and *metamodel protocol* – interface allowing to work with metamodel (consists of metamodel operations).

To create a metamodel, it is necessary to monitor base level system to find all known classes and to resolve their relationships and the way of their usage. A technique is required, which allows to watch and according to needs supplement existing code.

To extend base level application with analysis code we decided to use techniques of aspect-oriented programming (AOP). Aspect-oriented programming allowed modularization of crosscutting concerns. Besides this property,

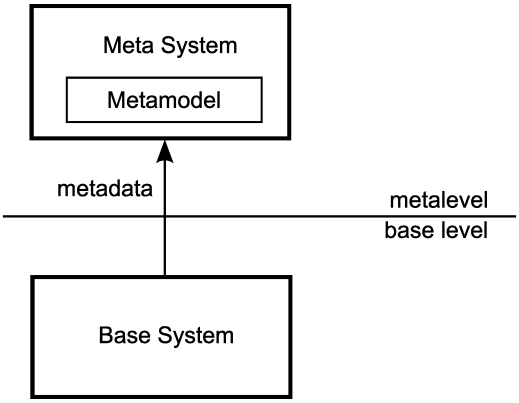

Figure 3: Base level and metalevel. Metamodel of usage of known classes is placed in metalevel.

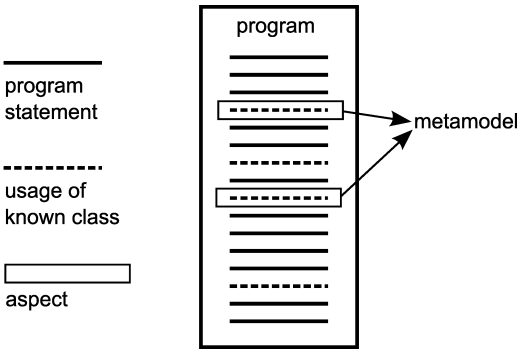

Figure 4: Metamodel creation using aspect-oriented technique. Usage of known classes are monitored in running program. The result information is used to build metamodel.

AOP allowed also extending existing code with a new functionality [8]. It is even possible to add a new functionality without access to source code.

With a help of aspect-oriented programming, the base system is extended with a new code, which has access to internal structures of the base system. While running the system, this code is gathering information about the system and on the basis of these information it is possible to build metamodel of specified concept of the base system (Fig. 4, 5).

Metamodel represents known state of selected concept of the base level system. Information about implementation in combination with knowledge base allows to create metamodel which describes the concept in higher level of abstraction than level of implementation. Created metamodel can be presented to user, who will later use it when applying software change. Metamodel contains only abstracted information about selected concept, avoiding implementation details.

3.2 Metamodel Change Reflection

Second part of described method is automatic change reflection. It is possible to affect the metamodel using metamodel operations from metamodel protocol. After invoking a metamodel operation, this will affect the metamodel, and this must be automatically reflected in base level system implementation (Fig. 6).

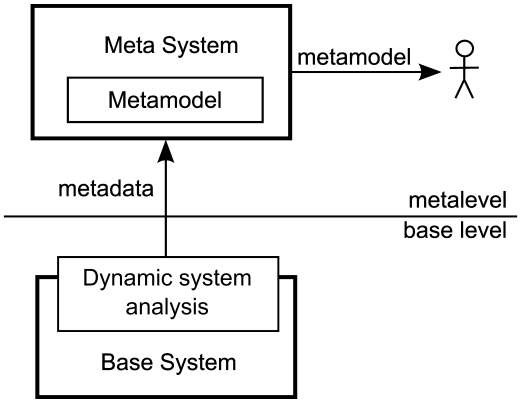

Figure 5: Base level system is enhanced with code for dynamic analysis. The results from this analysis is used to build metamodel in metalevel.

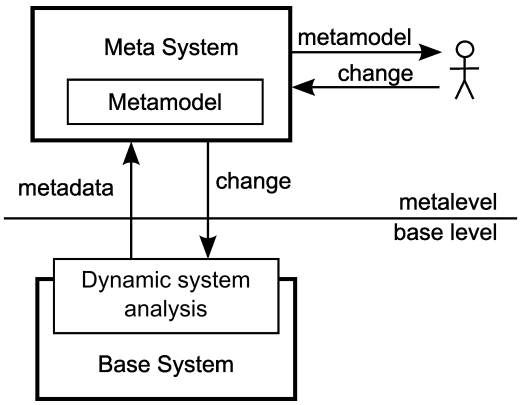

Figure 6: After the metamodel change, this change is automatically reflected in base level system.

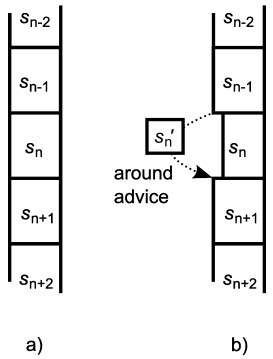

Figure 7: Original sequence of statements a) is adviced with new code b). This code avoids statement S_n and instead uses new statement S'_n

Base level implementation must be changed automatically by metasystem. The metasystem can realize the change according to information from knowledge base. Changing a general running program is a difficult task. Among other things, the biggest challenges are handling of active threads and transfer of program state. Replacing existing class with another one can break functionality of original program.

Proposed method uses an aspect-oriented technique also for application of change. With help of adding a new code to 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 new code implementing the change (Fig. 7). This way it is possible to apply certain group of changes without need of solving problems related to general dynamic software change.

The other possibility of changing behavior and properties of running object-oriented system is to change the system's object model. Access to system's internal structures makes it possible to use also this possibility. Therefore the task of the base level monitoring subsystem will be also to gather all needed information about internal structure of the system.

After changing the metamodel, change aspects are used to reflect the change in the base level application. The change is realized with help of around advice in combination with object model modifications.

3.3 Knowledge base

Essential part of the whole method is usage of knowledge base. Both described functions f and g uses the knowledge base to make transformation from one level of abstraction to another. The main task of the knowledge base is to replace a developer's knowledge during this transformation between different levels of abstractions.

The knowledge base must contain following information for each supported concept:

- *Implementation* – describes the way of implementation of the concept in the level of base system implementation. This information is used for tracking down the current state of the concept.
- *Model* – describes how the concept will be modeled in a metamodel (in higher level of abstraction).
- *Change* – describes how to reflect metamodel changes (in reaction on invocations of metamodel operations) in base (implementation) level.

Knowledge base will be manually filled with transformation functions, which will be individually prepared for each supported concept. Because of importance of knowledge base for the method, a quality of method will depend on size and quality of knowledge base.

3.4 Casual connection between base level and metalevel

Casual connection is a responsibility of aspect-oriented advices added to base level and the metasystem. From one point of view, a task of AOP advices is to gather all information about implementation of the concept in the base level. From this information, a metamodel is created (or updated).

When changing the metamodel, all modifications must be automatically reflected in the base level system. This will be done again with use of aspect-oriented programming and access to internal object structure of base level application.

4. Experiment of Metamodel Changes

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

Base level is presented by simple users management application. The application is controlled via standard graphical user interface. Besides standard management functions, the application allows to import list of users from external application.

Fig. 8 presents UML sequence diagram of process of importing users. The diagram was generated by Eclipse's Test & Performance Tools Platform (TPTP) Project. After reading data from network, these are parsed using instance of *java.util.Scanner* class. Each record occupies one line and consists of two values – user login (of type *String*) and user type (of type *Integer*).

Using the tool placed in metalevel, it is possible to automatically create metamodels of this network communication. First metamodel represents described format of communication (Fig. 9). To built this metamodel, the knowledge base had to be filled with details about using of class *java.util.Scanner* and the way of modeling it. In this case, it was enough to focus on context of invocations of methods *hasNext*, *next*, *nextInt*, etc.

Supported metamodel operations were adding new datatype and removing existing datatype. After change in communication format (required from the other peer), it is possible to adjust communication format of base level application. This can be simply done using provided metamodel operations. After metamodel change, this change is automatically (during runtime) reflected in the base level application – consequently new communication format is supported.

Second metamodel presents details about the way of communication (Fig. 10). The knowledge base was filled with information about different extensions of *java.io.InputStream* abstract class. Metamodel describes chain of instances of these classes used during communication. Metamodel operations allow adding new instances into this chain. This way it was possible to add new properties of communication, such as compression or encryption.

The last presented metamodel describes dialog flow within the application as a graph of states connected by transitions (Fig. 11). Transitions are presented as events invoked by button components. States describes visited dialogs of the application. After button event invocation, a dialog creation (or focus) is registered to record a transition to a new state. In this example, it was possible to remove states of the graph. After removing a state, component executing the transition to the state is removed.

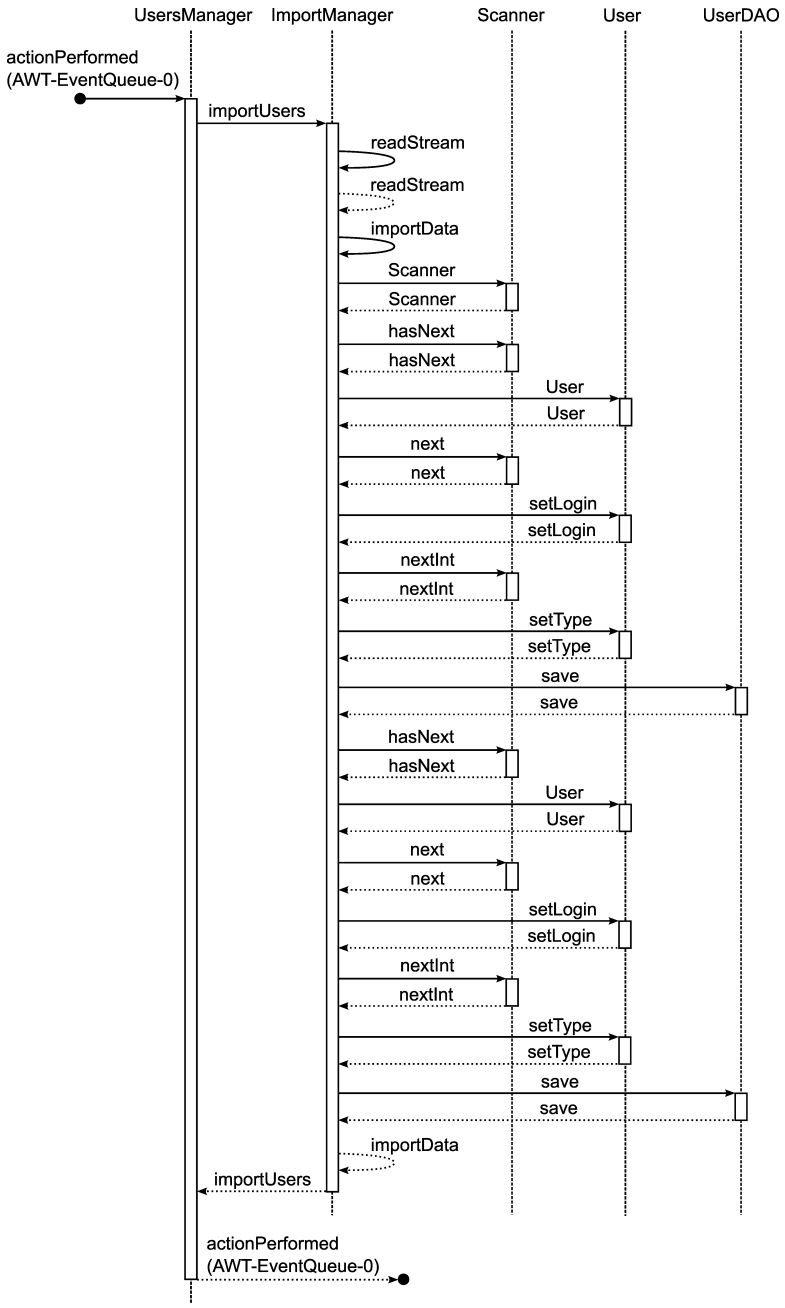

Figure 8: Sequence diagram of import users process.

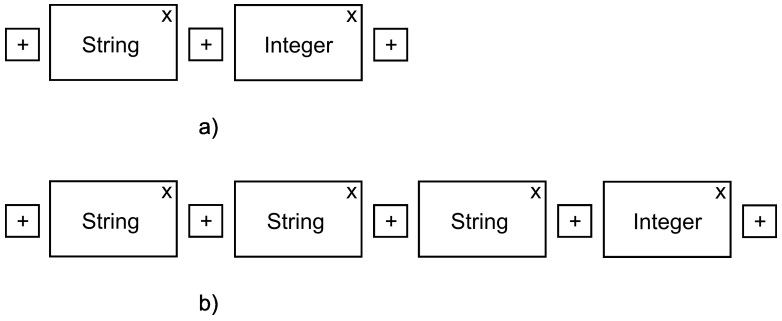

Figure 9: a) Metamodel of used format of communication. b) Metamodel after change – adding two new datatypes.

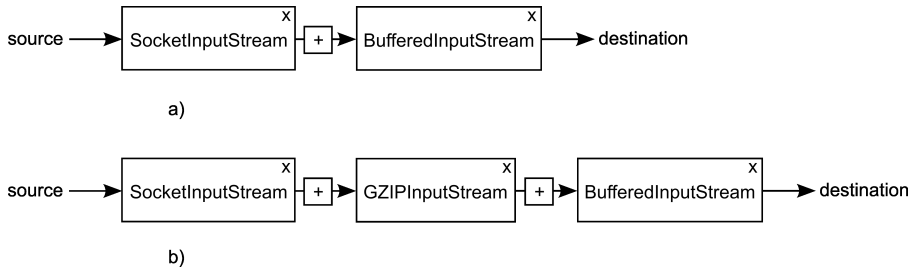

Figure 10: a) Metamodel of used way of communication. b) Metamodel after change – adding gzip compression.

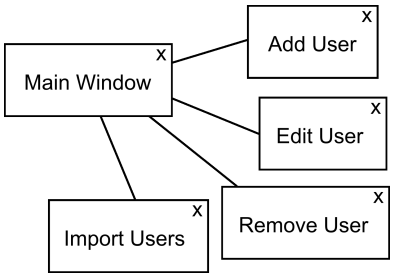

Figure 11: Metamodel of dialog flow.

Presented experiments used the knowledge base filled with few concepts implemented in Java Class Library. To model all possibilities of using this library is too complex. In real world projects, it is common to reuse huge amount of code – it will be helpful to prepare knowledge base for this reusable code, in order to improve its later comprehension and modifications.

5. Conclusions

The aim of our thesis was to contribute to the field of a program comprehension and evolution. We identified transition between different levels of abstraction as a way to improve possibilities in this field.

Our method allows automatic metamodel creation based on predefined knowledge base containing details about implementation and modeling. Based on this information, base level application is extended with a new code which is monitoring way of use of known classes. Extension with a new code is made possible using aspect-oriented techniques. Added code automatically tracks down all information about monitored concept implementation. According to this information, a metamodel representing concept is created in metalevel. The metamodel is automatically created in higher level of abstraction than implementation.

The knowledge base contains also information needed to reflect metamodel changes back in implementation level. After the metamodel change, second type of AOP advice is used to automatically reflect changes in base level application. The change is mostly implemented by AOP around advice, which allows to make execution of original code conditional. When needed, an original code is only extended, when needed it can be completely avoided.

Described experiment contains knowledge base filled with few concepts implemented in Java Class Library. The advantage of selection of these concepts is ability of using this knowledge base for any Java application (using compatible version of the library). The disadvantage is the complexity of the standard class library. A difficulty of projection between implementation level and metamodel in higher level of abstraction depends on specific concept.

Base level application needs no information about metalevel. It can be developed as a simple standalone application – all information related to metamodel creation are placed in metalevel. The presented tool allows applying changes during system runtime – there is no need to stop the base level application. Created metamodels describe only selected concept and hide implementation details, which simplifies its comprehension and modification.

Acknowledgements.

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

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37:72–82, May 1994.

[2] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[3] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, IWPC '00, pages 241–, Washington, DC, USA, 2000. IEEE Computer Society.

[4] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23, May 2000.

[5] M. M. Lehman, J. F. Ramil, and G. Kahen. 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.

[6] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. 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*, ASE '07, pages 234–243, New York, NY, USA, 2007. ACM.

[7] A. Olszak and B. N. Jørgensen. Remodularizing java programs for comprehension of features. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 19–26, New York, NY, USA, 2009. ACM.

[8] M. Oriol, W. Cazzola, S. Chiba, and G. Saake. Object-oriented technology. ecoop 2008 workshop reader. RAM-SE'08, chapter Getting Farther on Software Evolution via AOP and Reflection, pages 63–69. Springer-Verlag, Berlin, Heidelberg, 2009.

[9] D. Röthlisberger, O. Greevy, and O. Nierstrasz. 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, pages 79–100, New York, NY, USA, 2007. ACM.

[10] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *J. Syst. Softw.*, 49:3–15, December 1999.

[11] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7:49–62, January 1995.

Selected Papers by the Author

M. Forgáč, M. Vagač. Transformation of Functionality with Utilization of Metaprogramming and Reflection. In *Proceedings of the Eighth Scientific Conference of Young Researchers*, Košice: Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Košice, Slovakia, May 28, 2008, pp. 95–97, ISBN 978-80-553-0036-8.

M. Vagač, M. Forgáč. Aspects Mining by MetaLevel Construction. In *Proceedings of the Sixth International Symposium on Applied Machine Intelligence and Informatics*, Košice – Herľany: Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Košice, Slovakia, 2008, pp. 151–155, ISBN 978-1-4244-2106-0.

M. Vagač. Application Properties Abstraction Using AOP. In *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering*, High Tatras – Stará Lesná: Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Košice, Slovakia, 2008, pp. 141–146, ISBN 978-80-8086-092-9.

J. Kollár, J. Porub an, P. Václavík, M. Forgáč, M. Sabo, Ľ. Wassermann, F. Mrázik, M. Vagač, P. Klobušník. Software Evolution based on Software Language Engineering. *Computer Science and Technology Research Survey*, Košice, elfa, s.r.o., 2008, 3, pp. 25–30, ISBN 978-80-8086-100-1.

M. Vagač, J. Kollár. System Evolution by Metalevel Modification. In *Proceedings of the 10th International Conference on Engineering of Modern Electric Systems EMES'09*, Oradea, Faculty of Electrical Engineering and Information Technology of the University of Oradea, Oradea, Romania, 2009, Vol.: 2, Issue 1, pp. 71–74, ISSN 18446043.

M. Vagač, J. Siláči, J. Kollár. Metalevel Construction Using AOP. In *Proceedings of the Tenth International Conference on Informatics – INFORMATICS 2009*, Košice – Herľany: Slovak Society for Applied Cybernetics and Informatics, Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Košice, Slovakia, November 2009, pp. 178–183, ISBN 978-80-8086-126-1.

J. Kollár, Ľ. Wassermann, V. Vranič, M. Vagač. Reducing Structural Complexity of Software by Data Streams. In *INFOCOMP - Journal of Computer Science*, 8, 4, 2009, pp. 11–20, ISSN 1807-4545.

M. Vagač. System Aspect Mining from Java Library. In *Eleventh Scientific Conference of Young Researchers*, Košice: Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Košice, Slovakia, Accepted.

M. Vagač, J. Kollár. Improving Program Comprehension by Automatic Metamodel Abstraction. In *Computer Science and Information Systems*, Accepted.