Algorithms of the UML Class Diagram Analysis and Their Effectiveness: Strategy and Interface Insertion Transformations

Olga Deryugina[1, *] and *Evgeny* Nikulchev[1, 2]

[1]Moscow Technological University (MIREA), 119454 Moscow, Russia
[2]Moscow Technological Institute, 119334 Moscow, Russia

Abstract. A number of studies have investigated different means of the software refactoring. The question is whether there are any effective methods that could provide automated refactoring of the architecture on the design stage with no source code available yet. With the aim of evaluating this possibility, the algorithms of the UML class diagrams analysis were proposed. In addition, the computational experiment was conducted to evaluate the algorithms' performance depending on the initial diagram size.

1 Introduction

Model-Driven Architecture (MDA) approach [1], which was developed by the Object-Management Group (OMG), introduces a new software development paradigm. This paradigm proposes a design process, which starts with the creation of the PIM (Platform Independent Model). Then the PIM is automatically transformed into the PSM (Platform Specific Model).

MDA approach sets out a new research agenda of creating mathematical methods and software tools of model creation, transformation, storage and interchange.

Standards, developed to support the MDA conception, include MOF (Meta-Object Facilities) [2], UML (Unified Modelling Language) [3], XMI (XML Metadata Interchange) [4], etc.

One of the possible research challenges for MDA is the problem of model refactoring. The question is whether there are any effective methods that could provide automated refactoring of the architecture on the design stage with no source code available yet.

A number of studies have investigated different means of the model refactoring. There are two main approaches introduced by researchers.

The first one is based on the SBSE methods, which include GA, Simulated annealing, Swarm intelligence algorithms, etc. An algorithm receives a UML diagram and a fitness function as an input and gives a result diagram as an output [5-10].

The problem is that the result, maximizing the fitness function, can be unsatisfactory for the software designer. This can be solved by the interaction with the user on each step of the algorithm [11].

Software tools based on the SBSE approach include Dearthóir [12], Darwin [11], CODe-Imp [13], Bunch tool [14], etc.

That is why the second approach is connected with the developing frameworks of automated model refactoring, where the main role is given to the software designer. Framework analyses the model and gives recommendations for the refactoring aimed at the maximizing the fitness function, for example.

Software tools based on the second approach include AchE [15], UML Refactoring tool [16], etc.

In addition, various methods of UML class diagram transformation has been developed. The first one is a deterministic algorithm of transformation. However, the disadvantage of this approach is the lack of modifiability: the user cannot add new transformations to the database.

The second approach is usage of XML transformation facilities: XQuery [17] and XLST [18], for example. UML model should be exported to the XMI format, transformed and then imported from the XMI. The disadvantage is connected with the difficultness of understanding the transformations.

Nowadays the QVT (Query/View/Transformation) standard [19] has been developed by OMG. It defines a set of languages for model transformation. It gives the possibility of developing tools, which could allow users to add new transformation rules to the database.

The aim of this paper is to propose algorithms of UML class diagram analysis, which suggest Strategy and Interface transformations in order to minimize/maximize fitness function value.

The rest of this paper is organized as follows: First, the problem of the automated UML class diagram refactoring is formulated in Section 2. Then the UML Map abstract data structure, which stores elements of a UML class diagram in hash maps, is introduced in Section 3. In Section 4, the Strategy transformation analysis algorithm is proposed. In addition, the

* Corresponding author: o.a.derugina@yandex.ru

experimental results of applying this algorithm to class diagrams of different size are presented. After that, in Section 5 the Interface transformation analysis algorithm is proposed including the experimental results of applying it to class diagrams. In Section 6, the role of the proposed algorithms in the UML Refactoring Tool is introduced before concluding in Section 7.

2 The problem of the automated UML class diagram refactoring

UML class diagram transformation t can be described as the following mapping function:

$$t(d,E):d\to d', \quad (1)$$

where d is a UML class diagram, E is a set of diagram elements $e_i\in d, e_i\in\{C,R,I\}$, d' is a UML class diagram attained as a result of applying the transformation t to the diagram d.

Then the problem of the diagram refactoring can be formulated as searching such a set of pairs $\{t,E\}$, that:

$$d'=t(d,E),\Delta f(d')<0 \quad (2)$$

To be able to conduct the automated UML class diagram refactoring it is important to do the following:

1) formalize the semantics of UML class diagrams, which give an opportunity to check whether transformations keep the invariance of the semantic value;

2) formulate a refactoring criteria – fitness function $f(\cdot)$;

3) define a list of semantically equivalent transformations, which can be automatically performed by the refactoring algorithm.

The problem of the automated UML class diagram refactoring is connected with several difficulties.

First, it is hard to formulate a fitness function. Recent advances in UML class diagram quality evaluation provide a wide range of metrics [20–24]. However, it is hard to interpret most of these metrics. Scientists have conducted experiments aimed at evaluating the influence of different metrics (CBO, DIT or NOC [20], for example) on the level of the quality criteria [25–27]. So far, the results of these investigations sometimes are contradictory. In addition, the ways of measuring such quality criteria as the maintainability and understandability stay subjective.

Secondly, the way of searching parts of the diagram to be transformed depends on the transformation type: there is no general-purpose algorithm.

3 UML Map abstract data structure

To allow analysis and transformation of UML class diagrams the UML Map abstract data structure has been developed. It stores hash maps of the diagram's classes, interfaces and relations (see Fig. 1) to make diagram processing faster.

For example, the complexity of searching a class with the xmi:id = "X" in an XMI text document is evaluated as $O(n)$, while the complexity of searching in a hash map storing pairs of keys and values {xmi:id, Class} is evaluated as $O(1)$.

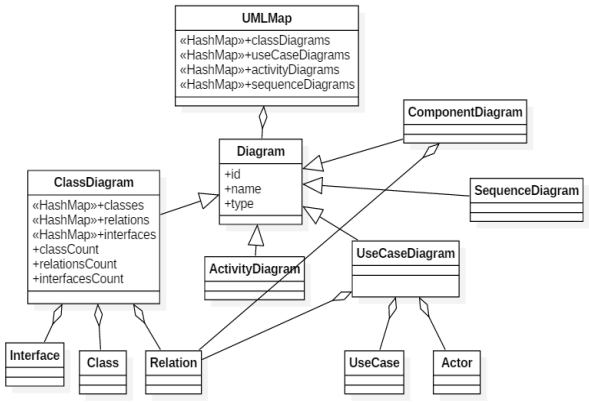

Fig. 1. Reduced class diagram of the UML Map abstract data structure

Special Translator has been developed to translate an XMI file to the UML Map and from the UML Map to an XMI file.

With the UML Map, it is possible to calculate metrics, search elements where transformation application is convenient, automatically transform diagrams, etc.

4 Strategy transformation analysis algorithm

The Strategy transformation means the encapsulation of several algorithms to provide their interchangeability (see Fig. 2 and Fig. 3).

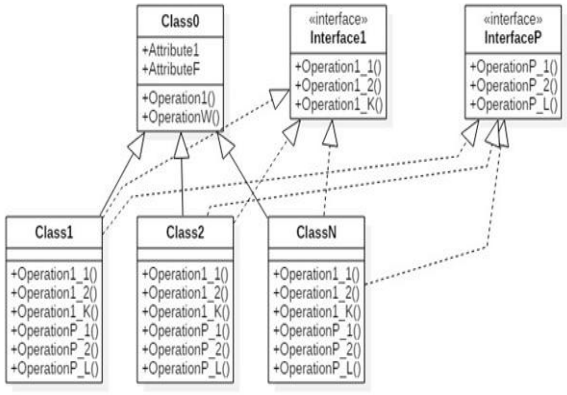

Fig. 2. Preconditions for the application of the Strategy transformation

Algorithm of searching sets of diagram elements E on which the Strategy transformation can be conducted can be described as follows:

1. For each class from the diagram d find those that have inheritors and add them to the hash map <Class, List<Class>> $l1=\{c_i,\{c_{i1},c_{i2},...\},...\}$.

2. For each class from the $l1$ check whether its inheritors implement any interfaces. If yes – add them to the hash map<Class,List<Class>> $l2=\{c_i,\{i_{i1},i_{i2},...\},...\}$.

3. For each class from $l2$ calculate $\Delta f(d)$. If $\Delta f(d)<0$ – add to the list $l3$ = {*parent_id*, {*child_classes_ids*}, {*interfaces_ids*}}.

4. Return $l3$.

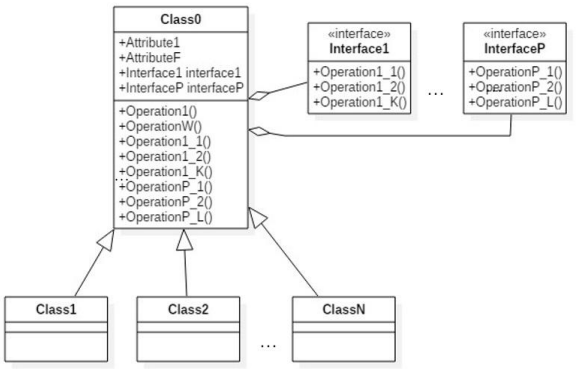

Fig. 3. The result of the Strategy transformation

For the abstract data structure UML Map, which stores hash maps of diagram classes, interfaces and relations, the algorithm can be described as follows:

```
1.for each c in classes
2.  inheritors = null
3.  for each r in relations
4.    if (r.getEndId() == c.getId())
5.      if (r.getType() == "generalization")
6.        inheritors.add(r.getStartId())
7.  l1.add(c.getId,inheritors)
8.for each c in l1
9.  interfaces = null
10.  for each r in relations
11.    if (r.getStartId() == c.getId())
12.      if (r.getType() == "realization")
13.        interfaces.add(r.getEndId())
14.  l2.add(c.getId, interfaces)
15.for each c in l2
16.  inheritors = l1.get(c.getId())
17.  interfaces = l2.get(c.getId())
18.  if (Δf(c,interfaces,inheritors,d)<0)
19.    l3.add(c,inheritors,interfaces)
20.return l3
```

A graph depicting the execution time of the algorithm depending on the UML class diagram size is shown in Fig. 4. The value of the execution time was calculated as the average execution time measured during 100 runs for each UML class diagram size. From the graph it follows that the algorithm time performance evaluation is $O(n)$.

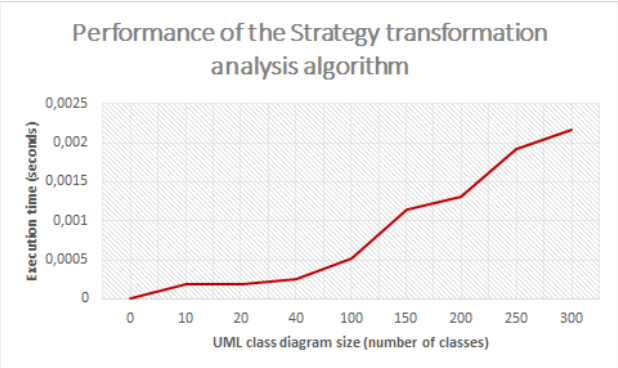

Fig. 4. Performance of the Strategy transformation analysis algorithm

5 Interface transformation analysis algorithm

The Interface insertion transformation involves the creation of an additional level between the classes, which realize a set of methods M, and the classes, which invoke these methods using dependency relations r_i.

Algorithm of searching sets of diagram elements E on which the Interface transformation can be conducted can be described for the abstract data structure UML Map as follows:

1. Find all classes $c_i \in d$, which do not implement any interfaces and add them to l_1.

2. Find among these classes c_i those that have dependency relations of the kind: $c_j \xrightarrow{dep} c_i$ and add them to l2.

3. For each c_i, calculate $\Delta f(d)$.

4. Return all c_i, where $\Delta f(d) < 0$.

For the abstract data structure UML Map the algorithm can be described as follows:

```
1.for each c in classes
2.  fl = false
3.  for each r in relations
4.    if (r.getStartId() == c.getId())
5.      if (r.getType()=="realization")
6.        fl = true
7.  if (!fl)
8.    l1.add(c)
9.for each c in l1
10.  fl = false
11.  for each r in relations
12.    if (r.getEndId() == c.getId())
13.      if (r.getType()=="dependency")
14.        fl = true
15.  if (fl)
16.    l2.add(c)
17.for each c in l2
18.  if (Δf(c,d)<0)
19.    l3.add(c)
20.return l3
```

A graph depicting the execution time of the algorithm depending on the UML class diagram size is shown in Fig. 5. The value of the execution time was calculated as the average execution time measured during 100 runs for each UML class diagram size. From the graph it follows that the algorithm time performance evaluation is $O(n)$.

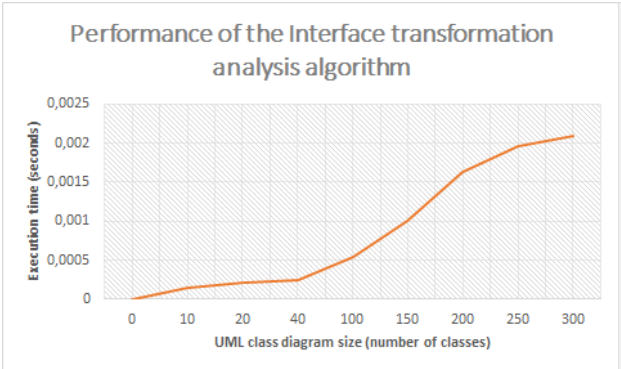

Fig. 5. Performance of the Interface transformation analysis algorithm

6 UML Refactoring tool

The introduced algorithms have become a part of the automated UML class diagram refactoring framework UML Refactoring Tool [16].

This tool allows analyzing UML class diagrams: it calculates various object-oriented metrics (such as Avg. DIT, Avg. CBO [20], DAC, SIZE2 [21], etc.) and proposes a list of transformations, which minimize a fitness function value. In addition, the tool applies the chosen transformation to the diagram. Framework can import/export projects from/to the XMI format.

7 Conclusions

The proposed UML Map abstract data structure makes it possible to analyze and transform UML class diagrams more convenient than in the XMI format.

The proposed algorithms of UML class diagram analysis allow searching elements of a UML class diagram, on which Strategy and Interface transformation can be applied in order to maximize/minimize fitness function value.

Time performance of the introduced algorithms has been evaluated with the help of the computational experiments. For both of them time performance evaluation is $O(n)$.

References

1. OMG Model Driven Architecture (http://www.omg.org/mda)
2. OMG Meta Object Facility Core Specification. Version 2.5.1. (http://www.omg.org/spec/MOF/2.5.1)
3. OMG Unified Modelling Language UML. Version 2.5 (http://www.omg.org/spec/UML/2.5)
4. XML Metadata Interchange (XMI) Specification. Version 2.4.2 (http://www.omg.org/spec/XMI/2.4.2)
5. M. Amoui, S. Mirarab, S. Ansari and C. Lucas, International Journal of Information Technology and Intelligent Computing, **1**, 235 (2006)
6. M. Bowman, L. Briand, and Y. Labiche, *Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms* (2002)
7. O. Räihä, *Genetic Synthesis of Software Architecture* (2008)
8. O. A. Deryugina, Russian Technological Journal, **1**, 26 (2015)
9. R. Lutz, JSA, **47**, 613 (2001)
10. C. Simons C., I. Parmee, *Single and multi-objective genetic operators in object-oriented conceptual software design*", (GECCO, 2007)
11. S. Hadaytullah, Vathsavayi, O. Räihä, and K. Koskimies, *Tool support for software architecture design with genetic algorithms* (IEEE CS Press, 2010)
12. M. O'Keeffe and M. Ó Cinnéide, *Towards automated design improvements through combinatorial optimization* (W2S, 2004)
13. M. O'Keeffe and M. Ó Cinnéide, *Search-based software maintenance* (CSMR, 2006)
14. S. Mancoridis, B.S. Mitchell, C. Rorres, Y.F. Chen, and E.R. Gansner, *Using automatic clustering to produce high-level system organizations of source code* (IWPC, 1998)
15. J. McGregor, F. Bachmann, L. Bass, and P. Bianco, *Using ArchE in the classroom: one experience* (CMU/SEI, 2007)
16. E. Nikulchev, O. Deryugina, International Journal of Advanced Computer Science and Applications, **7**(12), 76 (2016)
17. D. Chamberlin, IBM systems journal, **4**, 2002
18. XSL Transformations (XSLT) v1.0. W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt
19. Meta Object Facility (MOF) 2.0. Query/View/Transformation Specification. Version 1.3. http://www.omg.org/spec/QVT/1.3
20. S. R. Chidamber and C. F. Kemerer, IEEE Transaction on Software Engineering, **20**, 476 (1994)
21. W. Li and S. Henry*, Maintenance Metrics for the Object-Oriented Paradigm*" (ISMS, 1993)
22. B. Henderson-Sellers, *Object-oriented Metrics – Measures of Complexity* (New Jersey: Prentice-Hall, 1996)
23. M. Marchesi, *OOA Metrics for Unified Modelling Language* (Conference on Software Maintanance and Reengineering, 1998)
24. L. Briand, W. Devanbu and W. Melo, *An investigation into coupling measures for C++* (ICSE, 1997)
25. F. Brito e Abrue, W. Melo, *Evaluating the Impact of Object-Oriented Design on Software Quality* (International Metric Symposium1996)
26. M. Cartwright, *An Empirical View of Inheritance* (Information and Software Technology, 1998)
27. S. Chidamber, D. Darcy and C. Kemerer, IEEE Transactions on Software Engineering, **24**, 629 (1998)