UDC 004.415

METHOD AUTOMATED CLASS CONVERSION FOR COMPOSITION IMPLEMENTATION

Kungurtsev O. B. – PhD, Professor, Professor of the Software Engineering Department, Odessa Polytechnic National University, Odessa, Ukraine.

Bondar V. R. – Student of the Software Engineering Department, Odessa Polytechnic National University, Odessa, Ukraine.

Gratilova K. O. – Student of the Software Engineering Department, Odessa Polytechnic National University, Odessa, Ukraine.

Novikova N. O. – PhD, Associate Professor of the Department of Technical Cybernetics and Information Technologies named after professor R. V. Merct, Odessa National Maritime University, Odessa, Ukraine.

ABSTRACT

Context. Using the composition relation is one of the most effective and commonly used ways to specialize classes in object-oriented programming.

Objective. Problems arise when "redundant" attributes are detected in an inner class, which are not necessary for solving the tasks of a specialized class. To work with such attributes, the inner class has corresponding program methods, whose usage not only does not solve the tasks of the specialized class, but can lead to errors in its work. The purpose of this work is to remove "redundant" attributes from the inner class, as well as all methods of the class directly or indirectly (through other methods) using these attributes.

Method. A mathematical model of the inner class was developed, which allowed us to identify "redundant" elements of the class. The method of internal class transformation is proposed, which, based on the analysis of the class code, provides the developer with information to make a decision about "redundant" attributes, and then in the automated mode gradually removes and transforms the class elements.

Result. To approbate the proposed solutions, a software product Composition Converter was developed. Experiments were carried out to compare the conversion of classes in "manual" and automated modes. The results showed a multiple reduction of conversion time in the automated mode.

Conclusions. The proposed method of automated transformation of the inner class according to the tasks of the outer class when implementing composition allows to significantly reduce the time or the number of errors when editing the code of the inner class. The method can be used for various object-oriented languages.

KEYWORDS: object-oriented programming, classes, composition, syntactic analysis, class transformation.

ABBREVIATIONS

OOP – object-oriented programming.

NOMENCLATURE

attrName is a attribute identifier;

attrType is a attribute type;

cHead is a class header;

cHead1C2 is a new name of the inner class, reflecting the use in the outer class;

cName is a class name;

cName1 is a parent class name for cName (can be empty);

destr is a class destructor (if provided by the programming language);

fName is a method name;

mArgs is a set of method arguments;

mAttr is a set of class attributes;

mAttr1 is a set of attributes of class C1;

mAttr` is a a subset of the mAttr set containing redundant attributes;

mConstr is a set of class constructors;

mFunc is a set of ordinary methods of the class;

mMeth is a set of class methods;

mMeth1 is a set of methods of class C1;

mMeth11 is a methods of class C1 that are independent of mAttr`;

mMethR is a set of edited methods that have become independent of mAttr`;

mOperand is a set of operands:

mOperator is a set of method operators;

retType is a type of return value (empty for constructors and destructor).

INTRODUCTION

In object-oriented programming (OOP) there are two main ways of creating specialized classes based on existing ones – inheritance and extending the functionality of some class by using another class as an object attribute [1, 2]. Let us call the specialized class an outer class and the class of the included object an inner class. The object control of the inner class by an object of the outer class can be full and partial. In the first case the connection between the classes is called composition, and in the second case – aggregation. To implement

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14

composition, the outer and inner classes must have the following relations [3]:

– the inner class is a part of the outer class;

– the inner class can belong to only one outer class;

– the inner class (object) is controlled by the outer class (object);

– the inner class (object) does not know about the existence of the outer class (object).

Aggregation involves sharing of the inner class by several outer classes. In this case, conflicts of interests of outer classes may arise.

In practice, the use of composition is observed much more frequently than the use of aggregation. This work solves the problems associated with the composition usage. Composition has two significant advantages over inheritance [3]:

– allows adding additional functionality to the outer class with minimal changes in its structure;

– significantly reduces debugging time of the outer class, because the inner class is already ready-to-work.

The notion of a "ready-to-work class" requires explanation. If specializing the outer class by inheritance is understood as continuing to work on that class, then connecting the inner class involves searching for a suitable class from some library. By definition, the inner class in the vast majority of cases was not created for use in a particular outer class. To find a suitable inner class, candidates that provide the required functionality are considered. In many cases, a suitable candidate for the inner class has functionality beyond the required one. "Redundant functionality" consists of the existence of "redundant" methods and attributes. For example, in order to assign a bus to a driver to perform a trip, we can enter the attribute "Bus" into the class "Driver", which is a class. The "Bus" class may have many attributes and methods that model its engine, electrical system, running gear, repair information, etc., while the composition needs only the brand, registration number, number of seats, and possibly a few more attributes and corresponding methods. In case of the presence of "redundant" structural units in the inner class, the following problems arise [4]:

– when initializing a "redundant attribute", information is needed that is not defined by the task, which the outer class solves. This may be a source of initialisation errors;

– when working with an object of an inner class, it is possible to use methods directly or indirectly, which do not solve the tasks of the outer class, but introduce errors in their solution;

– methods that are "useful" from the point of view of tasks solved by the outer class may perform some actions on "redundant" attributes, which may also cause errors.

Thus, there is a problem of identifying, removing or "neutralizing" redundant attributes and methods in a class that is chosen as an inner class during composition.

According to the above problem, the following research tasks have been formulated:

– create a model of the inner class;

– develop a method to identify and remove "redundant" attributes and methods of the inner class, as well as to correct methods dependent on the deleted class elements.

1 PROBLEM STATEMENT

Suppose there is some program class c=<cHead, mAttr, mMeth>. When using this class as an object of another class *c2* (composition), a subset of attributes *mAttr`* turned out to be redundant. It is necessary to perform the transformation

$c \Rightarrow c1,$

where $c1 = < cHead1C2, mAttr1, mMeth1 >$.

Wherein $mAttr1 = mAttr \cap mAttr'$,

$mMeth1 = mMeth11 \cap mMethR$,

where $mMeth11 \in mMeth$,

$mMeth' = F(mAttr') \Rightarrow mMethR \neq F(mAttr')$.

2 REVIEW OF THE LITERATURE

Composition in programming languages is analyzed and applied at different levels. An attempt to develop a general approach to composition is made in work [5]. From our point of view, the recommendation to compose models for composition according to specific conditions is useful in this work.

In work [6], composition is considered at the level of language constructs of various domain-oriented languages. Of interest is a framework that allows creating a language from known constructs for a new subject area. Some principles of framework construction may find application to the present study.

In work [7] the principle of composition is applied at the level of individual operators and in [8] at the level of individual expressions, but it is also actually about making changes to programming language constructs rather than to program elements.

The conditions under which class-level composition has advantages compared to inheritance are described in sufficient detail in the literature [3,9], but the authors do not analyze the problems arising in its implementation.

In work [4] the composition problems are formulated, but the model is not developed. Therefore, the proposed solution is applicable only for a special case.

In work [10] it is proposed to allocate key classes for software understanding. This idea is relevant in the realization of composition if we represent "useful" attributes as key attributes. The authors did not formulate the task of any work with "redundant" classes (elements).

The issue of identifying and analyzing the effectiveness of class attributes is considered in work [11]. However, the study concerns only the attributes, the choice of which corresponds to the purpose of class creation, whereas in the conditions of composition the initial purpose of class application can be slightly changed.

In work [12], a class model is proposed, which represents its functionality quite completely, but does not provide for changes in the class.

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14

For the task of finding inheritance relations, an appropriate class model was created in [13]. The model provides class transformation by redistributing methods and attributes between classes, which is also applicable for this work, but does not allow identifying and removing "redundant" attributes and methods.

Class transformation is based on the extraction of certain constructs. Syntactic code analysis is considered in works [14, 15], where the main focus is on parser performance and creation of new convolution algorithms, whereas for this work the main requirement is the extraction of only certain code constructs.

A number of approaches to static code analysis [16] are applicable in the conditions of this work when "redundant" elements are used together with "useful" ones within one operator. An interesting proposal regarding combining static code analysis with object-oriented structure extraction is made in [17], but the authors do not offer an acceptable practical implementation of their project.

The work [18] shows the role of refactoring on the quality of object-oriented code. Accepting the recommendations of the authors of the work, the present study envisages not only checking the code for a given functionality, but also for compliance with design patterns [19].

3 MATERIALS AND METHODS

Class model.

Let's represent the class as a tuple:

$$c = <cHead, mAttr, mMeth>. \quad (1)$$

Let's represent the class header as a tuple:

$$cHead = <cName, cName1>. \quad (2)$$

Let's represent each attribute from the set *mAttr* as:

$$Attr = <attrName, attrType>. \quad (3)$$

Let's represent the set of methods as a tuple:

$$mMeth =<mFunc, mConstr, destr> . \quad (4)$$

Any element from mMeth has the form

$$mMeth_i = < fName_i, mArgs_i, retTipe_i, mOperator_i >. \quad (5)$$

Any operator is represented as a set of operands (variables, constants, function calls)

$$operator = mOperator.$$

Method of class transformation by removing redundant elements.

Initial data: some class *C*, which contains redundant attributes and methods from the point of view of its usage in composition.

Let's consider the case when the composite class is not inherited from another class.

First step. Let's analyze the set of attributes *mAttr* and form on its base the set of "redundant" attributes *mAttr'* and "useful" attributes *mAttr1*.

$$mAttr1 = mAttr \cap mAttr'.$$

Second step. Let's select from the set of all *mMeth* methods a subset of *mMeth'* methods, which use only attributes from the *mAttr'* set and do not use other methods of the same class (constructors and destructor are not analyzed yet).

$$mMeth' = \{ meth_i \mid mAttr1_k \notin_a meth_i \wedge meth_l \notin_m meth_i \}, i = 1, |mMeth|; k = 1, |mAttr'|; l = 1, |mMeth|,$$

where $\in_a$ and $\notin_a$ designate the use (non-use) of an attribute in a method; $\in_m$, $\notin_m$ – use (non-use) of other methods in this method.

Let's form a set of *mMeth1* methods that remain in the class:

$$mMeth1 = mMeth \cap mMeth'.$$

Third step. Let's select methods from the set *mMeth1* that do not use attributes from the set *mAttr1* and methods from the set *mMeth1*:

$$mMeth' = \{ meth_i \mid mAttr1_j \notin_a meth_i \wedge mMeth1_k \notin_m meth_i \}, i = 1, |mMeth1|; j = 1, |mAttr1|; k = 1, |mMeth1|.$$

Form the set of methods that remain in the class:

$$mMeth2 = mMeth1 \cap mMeth'.$$

Fourth step. Let's select from the set *mMeth2* a subset of methods that require editing (*mMethForAdjustment*). This category includes methods that contain "redundant" attributes and methods along with "useful" attributes and methods.

$$mMethForAdjustment = \{ meth_i \mid \exists\, mAttr'_j \in_a meth_i \vee \exists\, meth'_k \in_m meth_i \}, i = 1, |mMeth2|; k = 1, |mMeth'|; j = 1, |mAttr'|.$$

Fifth step. From each constructor the elements associated with redundant attributes are removed.

Let's represent the set of constructors in the form

$$mConstr = \{ constr_i \}, i = 1, |mConstr|.$$

Let's represent each constructor as a set of arguments and operators:

$$constr = \{ < mArgs, mOperator >\}.$$

The constructor operators should include not only operators in the body of the constructor, but also elements of the initialization list.

Operators that do not use the "useful" attributes *mAttr1* are defined

$$mOperator' = \{ operator_j \mid mAttr1_k \notin_{op} operator_j \}, j = 1, |mOperator|; k = 1, |mAttr1|,$$

where the relation $\notin_{op}$ – designates non-use of the attribute in the body of the operator.

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14

A new set of constructor operators is created

$$mOperator1 = mOperator \cap mOperator'.$$

Arguments that are used to initialize only redundant attributes are defined

$$mArgs' = \{args_j \mid args_j \notin_{op} mOperator1\},\ j = 1, |mArgs|.$$

A new set of the constructor arguments is created

$$mArgs1 = mArgs \cap mArgs'.$$

Let's select from the set of remaining constructor *mOperator1* operators a subset of operators that require correction (*mOperatorForAdjustment*). Operators that contain "redundant" attributes and arguments along with "useful" attributes and arguments should be included in this category.

$$mOperatorForAdjustment = \{\ operator_j \mid operator_j \in mOperator1 \wedge (\ \exists\ mAttr'_p \in_{op} operator_j\ )\},\ j = 1, |mOperator1|;\ p = 1, |mAttr'|;\ l = 1, |mArgs'|.$$

Sixth step (only for programming languages that use destructors). The elements associated with redundant attributes are removed from the destructor.

Let's represent the destructor as a set of operators

$$destr = mOperator.$$

Operators that do not use attributes from set *mMeth1* are defined

$$mOperator' = \{\ operator_j \mid mAttr1_l \notin_{op} operator_l\ \},\ j = 1, |mOperator|;\ l = 1, |mAttr1|.$$

A new set of destructor operators is created

$$mOperator1 = mOperator \cap mOperator'$$

Seventh step. The class C1 is formed based on *mAttr1, mMeth2*, transformed constructors and destructor.

$$c1 = <cHead1C2,\ mAttr1,\ mMeth1>,$$

where the new name *cHead1C2* indicates the modification of the original class C to conform to the requirements of class C2.

The case when an aggregate class is an inheritor of another class.

Option 1. There is a code of a parent class.

First step. The method proposed above is applied to the parent class.

Second step. The method proposed above is applied to the generated class.

Option 2. The code of the parent class is inaccessible.

First step. The "redundant" attributes introduced in the inherited class are determined. For them, the method proposed above is applied without performing the fifth, sixth and seventh steps.

Second step. The "redundant" attributes of the parent class are determined.

Third step. Methods that use only "redundant" attributes of the parent class are defined. Such methods should be made "neutral" depending on the context of their use. That is, such that their call does not lead to any changes in the context of their use.

A method of the parent class with a private method can be overridden. Then the call of the corresponding method of the parent class will be possible only when referring to the parent class.

The **fourth and subsequent steps** are performed according to the method proposed earlier.

4 EXPERIMENTS

In accordance with the proposed model and method of inner class transformation, a grammar is proposed that allows to extract from the class code the attribute description, the description of a regular method, the description of a constructor, the description of a destructor, an operator, an identifier and a method argument. In order to shorten the record of a number of rules widely used in grammars of programming languages, some right parts of definitions are omitted or replaced by an ellipsis

Grammar for highlighting necessary code elements:

```
$ class = {specifier} class class_name {specifier class_name} "{" {/ (description | operator /} "}"
$ description = type description_list";"
$ description list = description_item | description list "," description_item
$ type = standard_type | user_type
$ user_type = class_name | structure_name
$ standard_type = int | float | double | char |
      boolean | .....
$ identifier = ( letter | "_") { letter | number | "_"}
$ class_name = identifier
$ method_name = identifier
$ method = { specifier } method_name "(" argument list ")" "{"  { (description | operator /} } "}"
$ any_sequence_of_characters_without_";"
=............
$ operator= any_sequence_of_characters_without_; ";" | "{" operator "}"
```

The Composition Converter software product was created to implement the developed method. The scheme of the software product operation is shown in Fig.1. The Analyzer module allows you to select elements of the inner class in accordance with the given grammar. Command line compilers are used to check the correctness of the code of the edited methods. The scheme shows the sequential transformation of the original inner class $C \rightarrow C\_1 \rightarrow C\_2 \rightarrow C\_3$ by removing "redundant" methods and attributes, as well as editing methods for which "mechanical" removal of elements is impossible.

Fig. 2 shows a window with a list of attributes of the inner class, where the programmer can indicate redundant attributes.

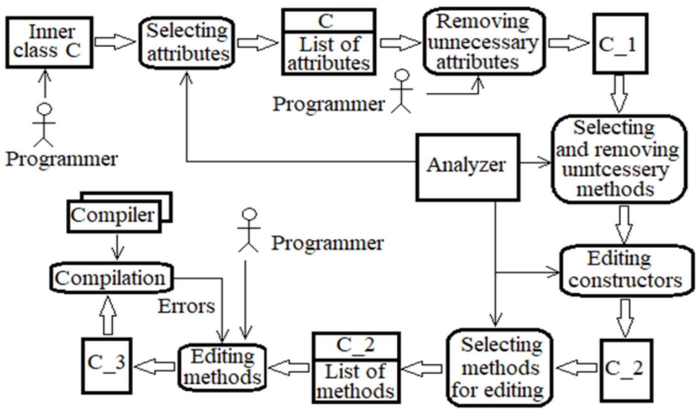
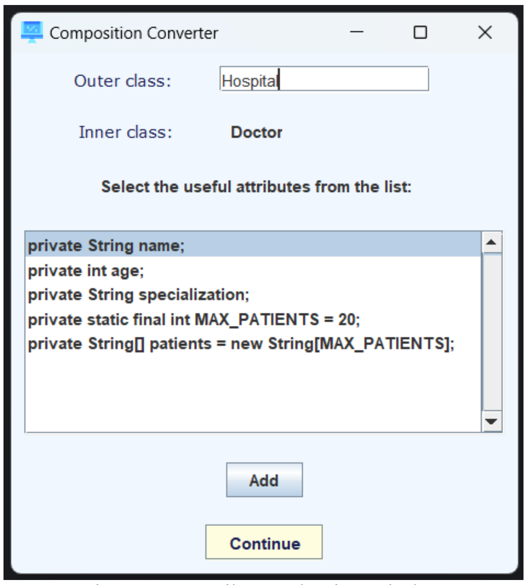


Figure 1 – Scheme of Composition Converter work



```
private String name;
private int age;
private String specialization;
private static final int MAX_PATIENTS = 20;
private String[] patients = new String[MAX_PATIENTS];
```

Figure 2 – Attribute selection window

5 RESULTS

A series of experiments were conducted to approbate the results of the study. The purpose of conducting the experiments was:

1. Verification of the quality of the program work.

2. Evaluation of the effectiveness of the proposed method.

In accordance with the first purpose, it was checked:

– identification of all cases of redundant constructions usage;

– deleting and automatic editing of constructions that do not require programmer intervention;

– providing the programmer with all constructions that require editing.

In accordance with the second purpose, it was determined:

– time for automated inner class transformation;

– time for “manual” inner class transformation.

For the study, 6 classes were developed from the subject areas “transport” and “health”. The number of attributes in the classes was 10, 20 and 30. The number of methods was 2–3 per attribute. 12 students from among the equally successful students of OO-programming subject were involved in the experiments. Each student performed conversion of 3 classes in “manual” mode and other 3 classes in automated mode using Composition Converter. Lists of “redundant” attributes were reported immediately before the experiments were performed.

No errors were found in the program operation at the stages of deletion and automatic transformation of class elements. Errors were observed when editing methods selected by the program. Errors in “manual” mode were observed at all stages.

In the “manual” mode, the time for class conversion ranged from 8 to 30 minutes. In the automated mode, the main time was spent on editing the methods allocated by the program and ranged from 2 to 10 minutes.

Fig. 3 shows the averaged data of the experimental results in the form of a graph, where *totalN* – is the total number of attributes, *unnecN* – is the number of redundant attributes, *mt* – is the time of “manual” class transformation, *at* – is the time of automated class transformation.

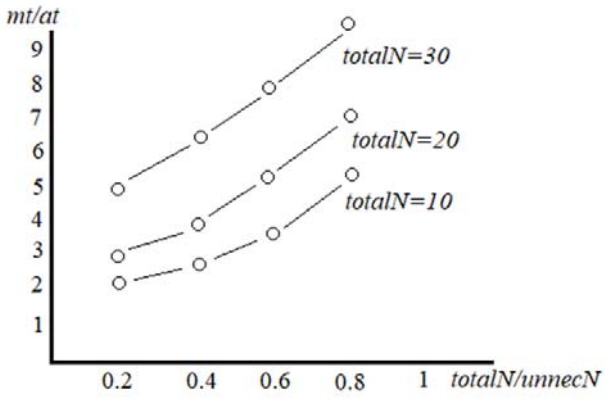


Figure 3 – Comparison of manual and automatic class transformation

6 DISCUSSION

Fig. 3 shows that the proposed transformation method proves to be effective for sufficiently large classes (10 or more attributes).

Errors during the “manual” class transformation were observed, but they were not counted because the result of the transformation was a valid code. Thus, the errors increased the transformation time.

The method leaves the programmer with the responsibility to edit functions (class methods) that use “useful” attributes along with “redundant” attributes. In general, it is extremely difficult to automate such editing, since it is determined by the tasks solved by the outer class and about which we have no information at the time of the research. Therefore, it was decided to limit to selecting such class methods and operators that use “redundant” attributes and loading them into the editor.

The proposed model and method are universal for most object-oriented programming languages with a high level of typing. However, the developed software product so far supports only Java and C++ languages.

The quality of editing class methods by the programmer is checked only for correct syntax, for which purpose command line compilers were connected to Composition Converter.

CONCLUSIONS

It is shown that the use of program classes as attributes of other classes in the implementation of composition is associated with significant problems caused by the presence of “redundant” attributes and “redundant” methods, the removal of which is a nontrivial task.

A mathematical model of the inner class is proposed, which allows us to consider a program class from the point of view of using its attributes, making it possible to formalize operations on class transformation.

A method is developed that allows automating the process of transforming an inner class, as a result of which all “redundant” attributes are removed from it, and all methods that use them are removed or edited.

Software has been created, which implements the proposed method of inner class transformation.

Approbation of the proposed solutions has shown their efficiency in the form of multiple reduction of time for class transformation (up to 10 times) in comparison with the existing technology (taking into account the time for error correction).

ACKNOWLEDGEMENTS

The work is supported by the state budget scientific research project of the Software Engineering Department, Odessa Polytechnic National University “Models, methods and tools of software engineering” (state registration number 0116U004528) and by of the Department of Technical Cybernetics and Information Technologies named after professor R.V. Merct, Odessa National Maritime University “Computer systems and information technology for the solution of applied problems” (state registration number 0123U101986).

We thank Alina I. Vitnova, who directly participated in the implementation of the class model.

REFERENCES

1. Forouzan B. A., Gilberg R. C++ Programming: An Object-Oriented Approach. McGraw-Hill Education, 2019, 960 p. https://www.booksfree.org/wp-content/uploads/2022/02/C-Programming-An-Object-Oriented-Approach-Behrouz-Forouzan.pdf
2. Lee G. Modern Programming: Object Oriented Programming and Best Practices. Packt Publishing, 2019, 266 p.
3. Kanjilal J. Composition vs. inheritance in OOP and C# [Electronic resource], InfoWord, 2023. Access mode: https://www.infoworld.com/article/3699129/composition-vs-inheritance-in-oop-and-c-sharp.html
4. Kungurtsev O., Bondar V., Gratilova K. Tranforming Classes for Composition Implementation, *Modern research in science and education: The 2nd International scientific and practical conference, Chicago, USA, 12–14 October 2023: proceedings.* Chicago, BoScience Publisher, 2023, pp. 143–148. ISBN 978-1-73981-123-5
5. Talcott C., Heinrich R., Duran F. et al. Composition of Languages, Models, and Analyses. New York, Springer, 2021, 311 p.
6. Kihlman L. Framework for Composition of Domain Specific Languages and the Effect of Composition on Re-use of Translation Rules: abstract of the dissertation … doctor of

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14

philosophy in computer science. Essex, University of Essex, 2021, 69 p.

7. Pfeiffer J., Rumpe B., Schmalzing D. et al. Composition operators for modeling languages: A literature review, *Journal of Computer Languages*, 2023, Vol. 76, P. 101226
8. Zhang W., Sun Y., Oliveira B. C. Compositional Programming, *ACM Transaction on Programming Lanquages and Systems*, 2021, Vol. 43, pp. 1–61 https://doi.org/10.1145/3460228
9. Nero R. Java inheritance vs. composition: How to choose [Electronic resource], InfoWord, 2020. Access mode: https://www.infoworld.com/article/3409071/java-challenger-7-debugging-java-inheritance.html
10. Wang L., Du X., Jiang B. et al. KEADA: Identifying Key Classes in Software Systems Using Dynamic Analysis and Entropy-Based Metrics, *PubMed*, 2022,Vol. 24, № 5, P. 652 DOI: 10.3390/e24050652
11. Rashidi H., Azadi F. On Attributes of Objects in Object-Oriented Software Analysis, *International Journal of Industrial Engineering & Production Research*, 2019, Vol. 30, pp. 341–352. DOI: 10.22068/ijiepr.30.3.341
12. Kungurtsev O., Novikova N., Reshetnyak M. et al. Method for defining conceptual classes in the description of use cases, *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*. Vilga, 6 November 2019, proceedings, SPIE P. 11176 doi: 10.1117/12.2537070
13. Kungurtsev O. B., Vytnova A. I. Determination of inheritance relations and restructuring of software class models in the process of developing information systems, *Radio Electronics, Computer Science, Control*, 2022, № 4(63), pp. 98–107.
14. Slivnik B., Mernik M. On Parsing Programming Languages with Turing-Complete Parser, *Mathematics,* 2023, Vol. 11, Issue 7. https://doi.org/10.3390/math11071594
15. Slivnik B. Context-sensitive parsing for programming languages, *Journal of Computer Languages*, 2022, Vol. 73, P. 101172. https://doi.org/10.1016/j.cola.2022.101172
16. Sudheer N., Hrushikesava S. Different Approach Analysis for Static Code in Software Development, *International Journal of Computer Sciences and Engineering*, 2016, Vol. 4 (9), pp. 111–118.
17. Wojszczyk R., Hapka A., Królikowski T. Performance analysis of extracting object structure from source code, *27th International Conference on Knowledge Based and Intelligent Information and Engineering Sytems (KES 2023), 2023 : proceedings*, *Procedia Computer Science,* 2023, Vol. 225, pp. 4065–4073. https://doi.org/10.1016/j.procs.2023.10.402
18. Kaur S., Singh P. How does object-oriented code refactoring influence software quality? Research landscape and challenges, *Journal of Systems and Software*, 2019, Vol. 157, P. 110394. https://doi.org/10.1016/j.jss.2019.110394
19. Wedyan F., Abufakher S. Impact of design patterns on software quality: a systematic literature review, *IET Software*, 2020, Vol. 14, Issue 1, pp. 1–17. https://doi.org/10.1049/iet-sen.2018.5446

Received 22.02.2024.
Accepted 02.04.2024.

УДК 004.415

МЕТОД АВТОМАТИЗОВАНОГО ПЕРЕТВОРЕННЯ КЛАСІВ ДЛЯ РЕАЛІЗАЦІЇ КОМПОЗИЦІЇ

Кунгурцев О. Б. – канд. техн. наук, професор кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», м. Одеса, Україна.

Бондар В. Р. – студентка кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», м. Одеса, Україна.

Гратілова К. О. – студентка кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», м. Одеса, Україна.

Новікова Н. О. – канд. техн. наук, доцент кафедри Технічна кібернетика й інформаційні технології ім. професора Р. В. Меркта Одеського національного морського університету, м. Одеса, Україна.

АНОТАЦІЯ

Актуальність. Використання відношення композиції – один із найефективніших і найчастіше використовуваних способів спеціалізації класів в об’єктно-орієнтованому програмуванні.

Мета роботи. Проблеми виникають при виявленні у внутрішньому класі зайвих атрибутів, які не потрібні для вирішення завдань спеціалізованого класу. Для роботи з такими атрибутами внутрішній клас має відповідні програмні методи, використання яких не тільки не вирішує завдання спеціалізованого класу, але й може призводити до помилок у його роботі. Метою роботи є видалення із внутрішнього класу «зайвих» атрибутів, і навіть всіх методів класу, які безпосередньо чи опосередковано (через інші методи) використовують ці атрибути.

Метод. Розроблено математичну модель внутрішнього класу, яка дозволила виділити «зайві» елементи класу. Запропоновано метод перетворення внутрішнього класу, який на основі аналізу коду класу надає розробнику інформацію для ухвалення рішення про «зайві» атрибути, а потім в автоматизованому режимі поетапно видаляє та перетворює елементи класу.

Результати. Для апробації запропонованих рішень розроблено програмний продукт Composition Converter. Проведено експерименти для порівняння перетворення класів у «ручному» та автоматизованому режимах. Результати показали багаторазове скорочення часу перетворення у автоматизованому режимі.

Висновки. Запропонований метод автоматизованого перетворення внутрішнього класу відповідно до завдань зовнішнього класу при реалізації композиції дозволяє суттєво скоротити час або кількість помилок при редагуванні коду внутрішнього класу. Метод може бути використаний для різних об’єктно-орієнтованих мов.

КЛЮЧОВІ СЛОВА: об’єктно-орієнтоване програмування, класи, композиція, синтаксичний аналіз, перетворення класу.

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14

ЛІТЕРАТУРА

1. Forouzan B. A. C++ Programming: An Object-Oriented Approach / B. A. Forouzan, R. Gilberg. – McGraw-Hill Education, 2019. – 960 p. https://www.booksfree.org/wp-content/uploads/2022/02/C-Programming-An-Object-Oriented-Approach-Behrouz-Forouzan.pdf
2. Lee G. Modern Programming: Object Oriented Programming and Best Practices. / G. Lee. – Packt Publishing, 2019. – 266 p.
3. Kanjilal J. Composition vs. inheritance in OOP and C# [Electronic resource] / J. Kanjilal. – InfoWord, 2023. – Access mode: https://www.infoworld.com/article/3699129/composition-vs-inheritance-in-oop-and-c-sharp.html
4. Kungurtsev O. Tranforming Classes for Composition Implementation / O. Kungurtsev, V. Bondar, K. Gratilova // Modern research in science and education: The 2nd International scientific and practical conference, Chicago, USA, 12–14 October 2023: proceedings. – Chicago : BoScience Publisher, 2023. – P. 143–148. ISBN 978-1-73981-123-5
5. Composition of Languages, Models, and Analyses / [C. Talcott, R. Heinrich, F. Duran et al.]. – New York : Springer, 2021. – 311 p.
6. Kihlman L. Framework for Composition of Domain Specific Languages and the Effect of Composition on Re-use of Translation Rules: abstract of the dissertation … doctor of philosophy in computer science / L. Kihlman. – Essex: University of Essex, 2021. – 69 p.
7. Composition operators for modeling languages: A literature review / [J. Pfeiffer, B. Rumpe, D. Schmalzing et al.] // Journal of Computer Languages. – 2023. – Vol. 76. – P. 101226
8. Zhang W. Compositional Programming / W. Zhang, Y. Sun, B. C. Oliveira // ACM Transaction on Programming Lanquages and Systems. – 2021. – Vol. 43. – P. 1–61. https://doi.org/10.1145/3460228
9. Nero R. Java inheritance vs. composition: How to choose [Electronic resource] / R. Nero. – InfoWord, 2020. – Access mode: https://www.infoworld.com/article/3409071/java-challenger-7-debugging-java-inheritance.html
10. KEADA: Identifying Key Classes in Software Systems Using Dynamic Analysis and Entropy-Based Metrics / [L. Wang, X. Du, B. Jiang et al.]. – PubMed. – 2022. – Vol. 24, № 5. – P. 652. DOI: 10.3390/e24050652
11. Rashidi H. On Attributes of Objects in Object-Oriented Software Analysis / H. Rashidi, F. Azadi // International Journal of Industrial Engineering & Production Research. – 2019. – Vol. 30. – P. 341–352. DOI: 10.22068/ijiepr.30.3.341
12. Method for defining conceptual classes in the description of use cases / [O. Kungurtsev, N. Novikova, M. Reshetnyak et al.] // Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments, Vilga, 6 November 2019: proceedings. – SPIE P. 11176 doi: 10.1117/12.2537070
13. Kungurtsev O. B. Determination of inheritance relations and restructuring of software class models in the process of developing information systems / O. B. Kungurtsev, A. I. Vytnova // Radio Electronics, Computer Science, Control. – 2022. – № 4(63). – P. 98–107.
14. Slivnik B. On Parsing Programming Languages with Turing-Complete Parser / B. Slivnik, M. Mernik // Mathematics. – 2023. – Vol. 11, Issue 7. https://doi.org/10.3390/math11071594
15. Slivnik B. Context-sensitive parsing for programming languages / B. Slivnik // Journal of Computer Languages. – 2022. – Vol. 73. – P. 101172. https://doi.org/10.1016/j.cola.2022.101172
16. Sudheer N. Different Approach Analysis for Static Code in Software Development / N. Sudheer, S. Hrushikesava // International Journal of Computer Sciences and Engineering. – 2016. – Vol. 4 (9). – P. 111–118.
17. Wojszczyk R. Performance analysis of extracting object structure from source code / R. Wojszczyk, A. Hapka, T. Królikowski // 27th International Conference on Knowledge Based and Intelligent Information and Engineering Sytems (KES 2023), 2023 : proceedings. – Procedia Computer Science, 2023. – Vol. 225. – P. 4065–4073. https://doi.org/10.1016/j.procs.2023.10.402
18. Kaur S. How does object-oriented code refactoring influence software quality? Research landscape and challenges / S. Kaur, P. Singh // Journal of Systems and Software. – 2019. – Vol. 157. – P. 110394. https://doi.org/10.1016/j.jss.2019.110394
19. Wedyan F. Impact of design patterns on software quality: a systematic literature review / F. Wedyan, S. Abufakher // IET Software. – 2020. – Vol. 14, Issue 1. – P. 1–17. https://doi.org/10.1049/iet-sen.2018.5446

© Kungurtsev O. B., Bondar V. R., Gratilova K. O., Novikova N. O., 2024
DOI 10.15588/1607-3274-2024-2-14