UDC 004.415.2

DETERMINATION OF INHERITANCE RELATIONS AND RESTRUCTURING OF SOFTWARE CLASS MODELS IN THE PROCESS OF DEVELOPING INFORMATION SYSTEMS

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

Vytnova A. I. – Student of the Software Engineering Department, Odessa Polytechnic National University, Odessa, Ukraine.

ABSTRACT

Context. The implementation of different use-cases may be performed by different development teams at different times. This results in a poorly structured code. The problem is exacerbated when developing medium and large projects in a short time.

Objective. Since inheritance is one of the effective ways to structure and improve the quality of code, the aim of the study is to determine possible inheritance relationships for a variety of class models.

Method. It is proposed to select from the entire set of classes representing the class model at a certain design stage, subsets for which a common parent class (in a particular case, an abstract class) is possible. To solve the problem, signs of the generality of classes have been formulated. The mathematical model of the conceptual class has been improved by including information about the responsibilities of the class, its methods and attributes. The connection of each class with the script items for which it is used has been established. A system of data types for class model elements is proposed. Description of class method signatures has been extended. A method for restructuring the class model, which involves 3 stages, has been developed. At the first stage, the proximity coefficients of classes are determined. At the second, subsets of possible child classes are created. At the third stage, an automated transformation of the class structure is performed, considering the identified inheritance relationships.

Results. A software product for conducting experiments to identify possible inheritance relationships depending on the number of classes and the degree of their similarity has been developed. The results of the conducted tests showed the effectiveness of the decisions made.

Conclusions. The method uses an algorithm for forming subsets of classes that can have one parent and an algorithm for automatically creating and converting classes to build a two-level class hierarchy. An experiment showed a threefold reduction in errors in detecting inheritance and a multiple reduction in time in comparison with the existing technology.

KEYWORDS: class model, class attribute, class method, data types, use case, inheritance.

ABBREVIATIONS

UC is a use-case;

OOP is an object-oriented programming;

OOA is an object-oriented analysis;

SP is a software product;

OOT is an object-oriented technologies.

NOMENCLATURE

Cp_j is a parent class;

C_{jq} is q-th descendant class that passed common attributes and methods to the parent class;

cHead is a class header;

mMeth is a set of functions (methods) of the class;

mAttr is a set of class attributes;

cName is a name of the class;

mResp is a set of class responsibilities;

uName is the name of the UC and the number of the point where the class was created, or a function was added to the class;

nP is a class responcibility, represented by a single phrase;

abstract is an abstract class;

cName1 is a name of the parent class for the *cName* class;

mChildCl is a set of child classes (filled only for an abstract class);

Numb is a number format;

Bool is a boolean value;

Text is any text;

Void is a function does not return the value;

NameS is a name of the type;

NameFi and Typei are the name and type of the *i*-th field;

NameL is a name of the type;

NameE is a name of the list element;

CPName is a type name (class name).

attrName is an identifier of the attribute;

attrResp is an attribute responsibility;

attrType is an attribute type;

fName is a name of the method;

fRespo is a responsibility of the method;

mRC_i is a set of class C_i responsibilities;

C_imRC_i is a number of class C_i responsibilities;

mRC_j is a set of class C_j responsibilities;

C_jmRC_j is a number of class C_j responsibilities;

mArgs is a set of method arguments;

returnVal is a function return value;

mRsArgs is a set of arguments that return the result of the calculation;

CA_i is an abstract class;

$mChildC_i$ is the set of its child classes;

CAS is the concatenation of the names of all classes that are included in the set $mChildC_r^{`}$ (hereinafter, the name is edited by the expert).

DOI 10.15588/1607-3274-2022-4-8

INTRODUCTION

The theory of OOP and OOA was elaborated in detail in the works of G. Booch and his colleagues [1] and continues to be developed and promoted [2, 3]. However, the practice of applying theoretical principles in the development of SPs faces many unsolved problems. The use of flexible technologies significantly speeds up the process of designing software products [4], however, it is possible to perform OOA to full extent only within the framework of the cascade model of the software life cycle [5]. In most OOT for creating software products functional requirements are written in the form of use cases (UC) [6]. UML is used to create UC diagrams, interaction diagrams, and class specifications. Stages of compiling the text of UC, class analysis, defining possible hierarchical relationships between them are not usually supported by design tools. The implementation of all main design stages within one iteration, which is typical for flexible technologies, allows carrying out a detailed OOA only for some fragments of the subject area. This creates a number of problems for the project [7], including defects in the architecture and structure of the class model. As a result, the program code requires detailed refactoring [8]. This is especially evident for medium and large projects, when teams of developers work in parallel to solve different problems (Fig. 1.). Under such conditions, there is a high probability that a possible "kinship" between classes will go unnoticed or will not cover all potential members of the hierarchy.

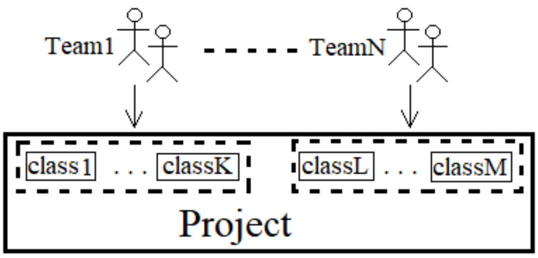

Figure 1 – Parallel development of the class structure

The purpose of the study is to select from the set of classes that represent the class model at a certain design stage, subsets for which a common parent class and automated restructuring of all classes related by inheritance relations are possible.

To achieve the stated goal, it is necessary to solve the following tasks:

– To formulate signs of class commonality;

– To improve the class model in order to provide comparison with other classes;

– To develop a method for restructuring the class model taking into account inheritance;

– To perform approbation of the research results.

1 PROBLEM STATEMENT

Let $mC = \{C_1, C_2, ..., C_i, ..., C_n\}$ be the set of class models of some software project. It is necessary to extract from mC such subsets of classes mC_1, mC_2, …, mC_k, for which common parent classes can be created. If some subset $mC_j = \{C_{j1}, C_{j2}, ..., C_{jq}\}$ is found, then it is transformed to the form $< Cp_j \{C'_{j1}, C'_{j2}, ..., C'_{jq}\} >$.

2 REVIEW OF THE LITERATURE

A good practical guide to inheritance is provided by [9], but it does not address the issue of inheritance of classes represented by models. In [10], it is proposed to put an abstract class as the basis of the hierarchy. It is shown that the effect of using an abstract class occurs when a number of subclasses are created on its basis in accordance with different specializations of the tasks being solved. However, the question of finding these specializations remains open. Disadvantages in the representation of classes in UML models are noted in [11]. The author suggests deepening your understanding of object-oriented concepts by determining relationships between actions and attributes, without considering the similarity of classes in terms of actions and attributes. In [12], the problem of the transition from the class model to the domain ontology is considered. An extension of the representation of classes, which, however, does not affect the identification of inheritance relations, is proposed.

In [13], the remodularization of object-oriented software systems is proposed, considering the connectivity, concatenation, index of the number and sizes of packages. The said principles of restructuring at the package level can be partly transferred to the class level.

The work [14] is devoted to the analysis of software quality at three levels. At the class level, it is proposed to introduce additional quality assessment metrics. However, they do not provide an assessment of the existing or possible hierarchical relationships between classes.

In [15], a two-level clustering of class models is proposed: at the level of semantics and structure. Obviously, this approach makes it possible to select "similar" classes. However, the analysis of the possible "kinship" between such classes was not performed in the work. A similar problem of determining groups of "close" classes was solved in [16]. But here the aim was to reduce testing resources, not to restructure classes.

The question of the comparative efficiency of manual and automated search for features of functions was considered in [17]. The idea of organizing the search for features not only in the code, but also in models is very productive.

The analysis of hierarchical relations of classes was performed in [18]. However, it is not the process of forming a hierarchical structure that is being studied, but its analysis for the purpose of preserving secret information in inherited methods.

In [19], a method for automated description of UC was proposed, which made it possible to further automate the process of building a model of conceptual classes [20]. At the same time, additional information about the connection of the class with the UC, methods and attributes of the class was placed in the model. Such a model [20] contains more information for searching for

DOI 10.15588/1607-3274-2022-4-8

class "kinship", but without significant development it cannot solve such a problem.

3 MATERIALS AND METHODS

Let start with an **improved model class**. In [20], a class model is proposed that can be taken as a basis. However, the specific task of finding a set of classes that can have a common "parent" requires a significant development of the said model. Let us formulate new requirements for the model:

– the class header is a comparison element. It must have the characteristic of responsibility.

– the class attribute is a comparison element. It must have the characteristic of responsibility and type;

– the class method is a comparison element. It must be represented by a responsibility and a signature;

– a class must have characteristics that define its role and relationships in the class hierarchy.

Basing on the foregoing, we will represent all the classes that are included into the project as a set:

$$mC = \{c\}, \tag{1}$$

and each class as a tuple:

$$c =< cHead, mMeth, mAttr > . \tag{2}$$

Now let's talk about a **class header**. To compare classes, it is proposed to introduce a set of responsibilities for which the class is used, formulated as separate sentences in the header of the class. In accordance with the technology of constructing a class model [21], a class is created when the UC "Create" item is implemented in the class model. At the same time, the first responsibility proposal is formed. For each subsequent point in the script, when the class must perform an action, a responsibility for the corresponding function, which is included into the set of class responsibilities is formed. For a possible tracing from the class model to the requirements (scenarios), the name of the corresponding UC and the number of the scenario item correspond to each new responsibility.

Further we will consider parent classes as abstract ones, since in our case they will not generally represent real objects of the subject area. Thus, the class header is represented as a tuple:

$$cHead =< cName, m\operatorname{Re}sp, inheri\tan ce > . \tag{3}$$

Each element of the set *mResp* is represented by a tuple

$$<uName,nP,r>. \tag{4}$$

An inheritance relationship is represented by a tuple:

$$<inheritTrait,mChildCl>, \tag{5}$$

where *inheritTrait* can take the following values: abstract, *cName1*, null (the class has no inheritance relationship with other classes), *mChildCl*.

In [20], a system of **data types** for a class model is proposed. In this work, this system has been developed at the expense of structured types.

Simple types: Numb, Bool, Text, Void.

Structured types. Struct – structure, in the general case, contains several fields of different types. The structure declaration has the following form:

$$Struct>NameS(n)(NameF1{:}Type1,NameF2{:}Type2, \ldots NameK{:}TypeK).$$

A List can represent a linear list, an array, a set, and so on.

The list declaration looks as:

$$List > NameL(NameE{:}Type).$$

The declaration of a reference to an object of the *CPType* class looks as:

$$CPType > CPName.$$

To provide the ability to compare **class attributes** it is proposed: to introduce the concept of the purpose (responsibility) of an attribute and data types.

As a result, each attribute from the set *mAttr* will be presented as:

$$Attr =< attrName, attr\operatorname{Re}sp, attrType > . \tag{6}$$

To provide the possibility of comparing **class methods**, it is proposed: for each method to formulate its obligation in the form of a short phrase, for instance, "calculation of the cost of the order"; for method arguments to use the rules formulated earlier for attributes.

As a result, each method from set mMeth(2) will take the form:

$$func =< fName, f\operatorname{Re}spo, mArgs, returnVal, mRsArgs > . \tag{7}$$

Figure 2 illustrates the resulting class model.

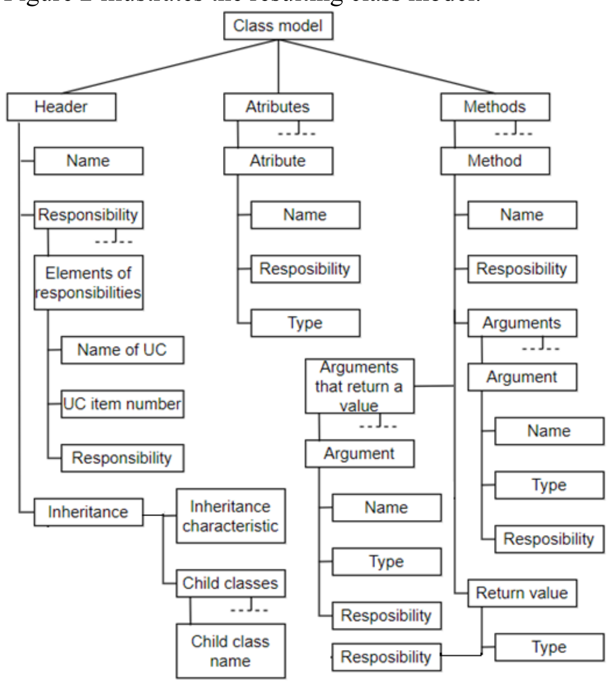

Figure 2 – The structure of the class model

DOI 10.15588/1607-3274-2022-4-8

The **class model restructuring method** involves four steps.

The first step is to determine the proximity of classes. Comparing two classes involves comparing class responsibilities, methods, and class attributes. To do this, it is necessary to compare various elements of the description of one class with other classes within the framework of the program class model, represented by a set *mC* (the total number of classes *nC*=|*mC*|).

For each comparison position, it is proposed to calculate the proximity coefficient

$$K = \frac{Number\_of\_matching\_elements}{Total\_number\_of\_elements}. \qquad (8)$$

When comparing the elements represented by the text, fuzzy string comparison functions were used [22]. Therefore, the result of the comparison will be a number not exceeding 1. A threshold value of the coefficient of proximity of responsibilities of the class *Kcmin* has been introduced, below which it makes no sense to search for the "kinship" of classes.

To compare the responsibilities of classes, we transform the set of responsibilities $mResp_i$ (3) of a certain class C_i, excluding references to the UC and the scenario item.

$$m\mathrm{Re}sp_i \Rightarrow mRC_i, \qquad (9)$$

where $C_i mRC_i = \{r_{i,1},...,r_{i,n}\}$, $C_i nRC_i =| mRC_i |$, $mRC_j = \{r_{j,1},...,r_{j,m}\}$ and $C_j mRC_j = \{r_{j,1},...,r_{j,n}\}$, $nRC_j =| mRC_j |$.

Let us define a set of overlapping responsibilities of classes C_i and C_j

$$mRC_{i,j} = \{ro_q \mid ro_q = r_{i,j} \wedge ro_q = r_{j,p}\}, \qquad (10)$$

and their number

$$nRC_{i,j} =| mRC_{i,j} |. \qquad (11)$$

If $nRC_{i,j} = 0$, then class comparison stops.

When comparing class methods, we proceed from the following considerations. Each time when a class is used to implement a script item, a responsibility is added to the class header. The same responsibility is attributed to the class function that implements it in the script item. To determine the identity of two functions with overlapping responsibilities from classes C_i and C_j, to match of all elements from (7) except the function names is required. Let us represent the set of coinciding functions of classes C_i and C_j in the form $mMethC_{i,j}$. If no match is found for a pair of functions, then $nRC_{i,j}$ is reduced by one.

Match of class attributes does not affect the assessment of class proximity degree, because there are methods that do not use the attributes of their class. However, matching attributes must be identified for further class transformation. To determine the identity of two attributes from classes C_i and C_j, their types and responsibilities must match. Let us represent the set of matching attributes of classes C_i and C_j in the form $mAttrC_{i,j}$.

The result of comparing two classes is called the proximity coefficient of the said classes *ER*. Its value must be different for classes C_i and C_j. For a class C_i:

$$ER_{i,j} = \frac{nRC_{i,j}}{nRC_i}. \qquad (12)$$

For a class C_j:

$$ER_{j,i} = \frac{nRC_{i,j}}{nRC_j}. \qquad (13)$$

The overall coefficient:

$$ERO_{i,j} = \frac{ER_{i,j} + ER_{j,i}}{2}. \qquad (14)$$

The second stage is the construction of the class proximity matrix. To identify the possible "kinship" of classes from set *mC*(1), it is proposed to use the matrix of class proximity. An example of such a matrix is presented in Table 1.

Table 1 – Matrix of class proximity

Classes	C1	C2	C3	C4	C5	C6	C7	C8	C9
C1	X	0	$ER_{1,3}$	0	0	$ER_{1,6}$	0	$ER_{1,8}$	0
C2	0	X	$ER_{2,3}$	0	$ER_{2,5}$	$ER_{2,6}$	0	0	0
C3	$ER_{3,1}$	$ER_{3,2}$	X	$ER_{3,4}$	0	0	0	0	$ER_{3,9}$
C4	0	0	$ER_{4,3}$	X	$ER_{4,5}$	$ER_{4,6}$	0	0	0
C5	0	$ER_{5,2}$	0	$ER_{5,4}$	X	0	0	0	0
C6	$ER_{6,1}$	$ER_{6,2}$	0	$ER_{6,4}$	0	X	0	$ER_{6,8}$	0
C7	0	0	0	0	0	0	X	0	0
C8	$ER_{8,1}$	0	0	0	0	$ER_{8,6}$	0	X	$ER_{8,9}$
C9	0	0	$ER_{9,3}$	0	0	0	0	$ER_{9,8}$	X

DOI 10.15588/1607-3274-2022-4-8

The cells of the matrix contain the values of the proximity coefficients for all pairs of classes from set mC. For instance, it follows from the matrix that there is no commonality between classes C_1 and C_2, but there is a commonality between classes C_2 and C_5.

The presence of commonality between class C_1 and classes C_3, C_6, C_8 does not mean that there will be one parent class for all these classes. The search for the optimal solution will consist in the fact that for any class from group C_3, C_6, C_8, the condition for combining with C_1 is the greatest value of proximity with this particular class. For example, C_6 will enter a group with C_1 if $ER_{1,6} = \max(ER_{6,2}, ER_{6,4}, ER_{6,8})$.

The third stage is the formation of a set of abstract classes. At this stage, as a result of processing the matrix, it is necessary to form a set of abstract (parent) classes mCA (initially, the set is empty), each element of which has the form

$$mCA_i = < CA_i, mChildC_i > . \tag{15}$$

Previously, we will place classes that can potentially become child classes in the set of child classes. Let us denote such a set $mChildC^{'}$. The sequence of operations for the formation of the said sets is represented by the algorithm for identifying parent (abstract) classes:

1. To define the set of all classes and the set mC of abstract classes c.

2. To fill in the generality matrix of the size $K \times K$, where $K = | mC |$. To set the matrix row index i=1 and the abstract class index r=1.

3. For each proximity coefficient $ER_{j,n} \neq 0$, to calculate the total proximity coefficients $ERO_{j,n}$ for $j = i+1, K$.

4. If some $ERO_{j,n} > ERO_{i,n}$ is found, then $ERO_{i,n}$ is reset to zero. Otherwise, all $ERO_{j,n}$ are set to zero. If there is no more than one ERO in the current line, then go to step 6.

5. The set $mChildC^{'}_r$ contains all classes of the i-th row for which $ER_{i,n} \neq 0$. Only the name of the abstract class is entered as CA_r. $cName$=CAS. To increase index r by 1.

6. To increase index i by 1. If i<K, go to step 3.

7. Completion of the algorithm.

The fourth stage is the formation of parent (abstract) and child classes. For each abstract class with the name CAS_r, it is necessary to form a header, methods and attributes using a set $mChildC^{'}_r$ of classes. Each class in the set $mChildC^{'}_r$ must be converted into a derived class CAS_r by changing the header, excluding methods and attributes that passed into CAS_r.

The solution to this problem is formulated as a class restructuring algorithm:

1. We determine the possible number of abstract classes Ka=|mCA| and set the index of the first abstract class i=1.

2. We determine the number of possible child classes for the i-th abstract from $Kc_i = | mCA_i.mChildC^{'} |$ and define the responsibilities $m\mathrm{Re}sp_i$ of an abstract class CAS_i by identifying, in accordance with (10), the general responsibilities of classes from $mChildC^{'}_i$. We write in the inheritance relation $inheritTrait_i = abstract$, in the set $mChildCl_i$ we write the names of classes from $mChildC^{'}_i$.

3. We determine methods $mMethCA_i$ of an abstract class CAS_i by identifying common methods of classes from $mChildC^{'}_i$.

4. We determine the attributes $mAttrCA_i$ of an abstract class CAS_i by identifying common attributes of classes from $mChildC^{'}_i$.

5. We set the index of the child class j=1.

6. In the class header $c_{i,j} \in mCA_i.mChildC^{'}_j$, we set the inheritance flag $c_j.cHead.inheritTrait = CAS_j$.

7. We remove methods of class CAS_i $c_{i,j}$ from the class $c_{i,j}.mMeth := c_{i,j}.mMeth \cap CAS_i.mMeth$.

8. We remove attributes of class CAS_i $c_{i,j}$ from the class $c_{i,j}.mAttr := c_{i,j}.mAttr \cap CAS_i.mAttr$.

9. We set j:=j+1. If $j <= Kc_i$, then go to step 6. Otherwise, go to step 7.

10. We demonstrate the analytics of the abstract class CAS_r and its child classes $mCA_i.mChildC`$. If inheritance is asserted, then each class from mC for which $mC.cName = mChildC_{i,j}.cName$ is replaced by the corresponding class $mChildC_{i,j}$ and an abstract class named CAS_i is added to the set mC.

11. We set i:=i+1. If i<=Ka, then go to step 2. Otherwise, finish the algorithm.

4 EXPERIMENTS

In accordance with [21], a simplified scheme for constructing a class model is shown in Fig. 3.

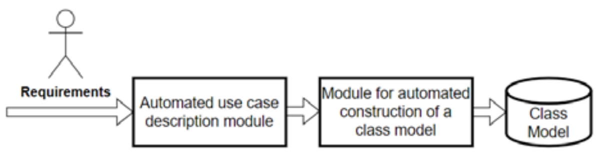

Figure 3 – Simplified scheme for building a class model

DOI 10.15588/1607-3274-2022-4-8

The system analyst, basing on the analysis of the subject area and consultations with an expert describes the UC using the UseCaseEditor program [20]. Basing on the obtained UCs, a programmer (perhaps a system analyst) creates a class model using the ModelEditor program [21].

To apply the proposed method of restructuring the class model, a software product HeirClass+ was developed.

Within the framework shown in Fig. 3 the technology (working mode), it is difficult to test the method of searching for inheritance relations, since it is impossible to select such UCs that would provide many classes in the model with the necessary characteristics in advance. Therefore, to test the decisions made, a software module was developed that allows you to create a class model bypassing the stage of automated UC description (experimental mode). Fig. 4 shows the class model restructuring scheme in an experimental and operational modes.

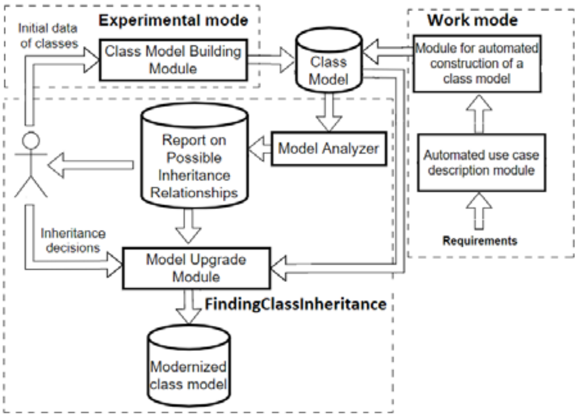

Figure 4 – Scheme for testing the decisions made and testing modules

For performing experiments 15 programmers (3rd year students) were involved. Of these, 5 teams were formed. For each team, requirements to 4 classes were formulated in the following form: "The class must perform .... The class contains a method that, basing on ..., returns .... The class contains an attribute that represents…". The requirements were distributed in such a way that one team could not be given the task of describing potentially related classes. It was supposed that, in accordance with the requirements, there could be 6 groups of "related" classes.

5 RESULTS

After completing the work on the models, the participants in the experiment were asked to identify potential inheritance relationships in a variety of classes. Simultaneously the program HeirClass+ with identical source data was started. After 2 hours, the teams identified 8 class groups with signs of inheritance relationships out of 9 supposed ones. Of these, 4 groups were accepted for restructuring. Program HeirClass+ identified 8 groups within 10 seconds. Of these, 5 groups were accepted for restructuring at a threshold commonality rate of 35%. In addition, HeirClass+ performed the restructuring flawlessly.

Table 2 shows the matrix of generality for the first 10 classes, obtained on the basis of the work of the program HeirClass+(classes named C1-C10 for brevity).

Figure 5 presents a piece of information that is offered to the developer for deciding about inheritance.

Table 2 – Class commonality matrix (experiment)

Classes	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
C1	X	0%	29%	0%	43%	57%	29%	29%	43%	14%
C2	0%	X	38%	0%	38%	13%	0%	38%	13%	50%
C3	20%	30%	X	10%	10%	30%	0%	50%	0%	40%
C4	0%	0%	17%	X	0%	0%	33%	0%	17%	0%
C5	38%	38%	13%	0%	X	13%	13%	25%	50%	63%
C6	14%	57%	43%	0%	14%	X	0%	43%	14%	14%
C7	20%	0%	0%	20%	10%	0%	X	0%	20%	0%
C8	29%	43%	71%	0%	29%	43%	0%	X	14%	29%
C9	30%	10%	0%	10%	40%	10%	20%	10%	X	20%
C10	13%	50%	50%	0%	63%	13%	0%	25%	25%	X

DOI 10.15588/1607-3274-2022-4-8

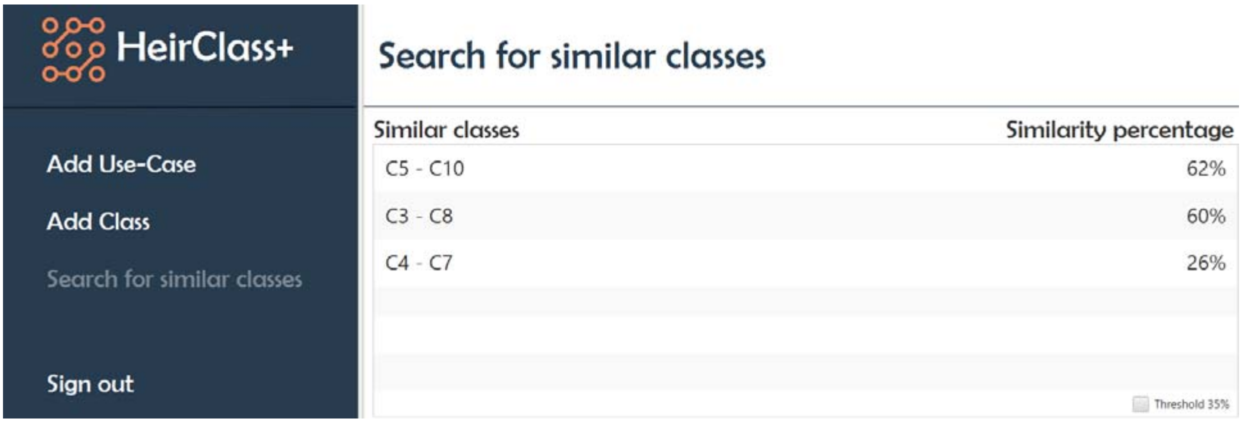

Figure 5 – Class comparison result

6 DISCUSSION

Until now, class inheritance has been studied in terms of analyzing the effectiveness of its application [10], building class libraries, developing conditions and recommendations for specializing generated classes [18]. In this work, for the first time, the problem of automated search for possible inheritance relations and their implementation for a set of classes is solved. Class conversion automation is used in refactoring [8]. However, for refactoring, the object of modernization is the code, and the operations are initiated by a specialist.

From what has been said, it follows that the proposed method can only be compared with "manual" processing of a set of classes. The experiment showed that automated analysis was performed hundreds of times faster than manual analysis with a significant reduction in the number of errors, and class conversion turned out to be error-free.

CONCLUSIONS

It is shown that modern iterative software development technologies lead to the creation of a poorly structured code, which requires refactoring at relatively late stages of software design and is associated with high costs.

The paper solves the problem of automated determination of inheritance relations for a set of classes. For this purpose, signs of the generality of classes have been formulated; the class model has been improved by defining the concept of responsibility class, method, attribute; detailed description of the method signature has been given; a data type system for the class model has been proposed.

A method for restructuring the class model has been developed. The method uses an algorithm for forming subsets of classes that can have one parent and an algorithm for automatically creating and converting classes to build a two-level class hierarchy.

The results of the study are implemented in the HeirClass+ software product. An experiment using HeirClass+ showed a threefold reduction in errors in detecting inheritance and a multiple reduction in time in comparison with the existing technology.

REFERENCES

1. Booch G., Maksimchuk R. A., Engle M. W., Young B. J., Conallen J., Houston K. A., Wesley A. Object-Oriented Analysis and Design with Applications 3rd Edition. Boston, Addison-Wesley Professional, 2007, 694 p.
2. Lee G. Modern Programming: Object Oriented Programming and Best Practices. Birmingham, Packt, 2019, 266 p.
3. Baesens B., Backiel A., Broucke S. Beginning Java Programming: The Object-Oriented Approach. Birmingham, Wrox, 2015, 672 p.
4. Brand M., Tiberius V., Bican P. M., Brem A. Agility as an innovation driver: towards an agile front end of innovation framework. Potsdam, Springer, 2021, pp. 157–187.
5. Adeagbo M. A., Akinsola J., Awoseyi A. A., Kasali F. Project Implementation Decision Using Software Devel-opment Life Cycle Models: A Comparative Approach, *Journal of Computer Science and Its Application,* 2021, No. 28, pp. 122–133.
6. Jacobson I., Spence I., Bittner K. USE-CASE 2.0 The Guide to Succeeding with Use Cases [Electronic Recourse]. Access mode: https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf
7. Arcos-Medina G., Mauricio D. The Influence of the Application of Agile Practices in Software Quality Based on ISO/IEC 25010 Standard, *International Journal of Information Technologies and Systems Approach,* 2020, №13, pp. 1–27.
8. Mohan M., Greer D. A survey of search-based refactoring for software maintenance, *Journal of Software Engineering Research and Development*, 2018, №6, pp. 1–52.
9. Ryan M. Mastering OOP: A Practical Guide to Inheritance, Interfaces, and Abstract Classes [Electronic Recourse]. Access mode: https://www.smashingmagazine.com/2019/11/guide-oop-inheritance-interfaces-abstract-classes/
10. Taubler D. When to Use Abstract Classes [Electronic Recourse]. Access mode: https://betterprogramming.pub/when-to-use-abstract-classes-70fe526165ac
11. AI-Fedaghi S. Classes in Object-Oriented Modeling (UML): Further Understanding and Abstraction, *International Journal of Computer Science and Network Security,* 2021, №21. pp. 139–150.
12. Minh Hoang Lien Vo, Hoang Q. Transformation of UML class diagram into OWL Ontology, *Journal of Information and Telecommunication*, 2020, No. 4, Issue 1.

DOI 10.15588/1607-3274-2022-4-8

13. Gandhi P., Pradeep K. Optimization of Object-Oriented Design using Coupling Metrics, *International Journal of Computer Applications*, 2011, No. 27, pp. 41–44.
14. Saeed M. G., Alasaady M. T. Three Levels Quality Analysis Tool for Object Oriented Programming, *International Journal of Advanced Computer Science and Applications*, 2018, № 9, pp. 522–536.
15. Zongmin Ma, Zhongchen Yuan, Yan Li Two-level clustering of UML class diagrams based on semantics and structure, *Information and Software Technology*, 2021, No. 130, 106456.
16. Miao Zhang, Jacky Wai Keung, Yan Xiao, Md Alamgir Kabir Evaluating the effects of similar-class combination on class integration test order generation, *Information and Software Technology*, 2021, №129, 106438.
17. Pérez F., Echeverría J., Lapeña R., Cetina C. Comparing manual and automated feature location in conceptual models: A Controlled experiment, *Information and Software Technology*, 2020, No. 125, 106337.
18. Benlhachmi K., Benattou M. A Formal Model of Conformity and Security Testing of Inheritance for Object Oriented Constraint Programming, *Journal of Information Security,* 2013, №4, pp. 113–123.
19. Vozovikov Yu. N., Kungurtsev A. B., Novikova N. A. Information technology for automated compilation of use cases, *Science practices of Donetsk National Technical University*, 2017, No. 1 (30), pp. 46–59.
20. Kungurtsev O., Novikova N., Reshetnyak M., Cherepinina Ya., Gromaszek K., Jarykbassov D. Method for defining conceptual classes in the description of use cases. Odessa: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019, 2019, 1117624.
21. Kungurtsev O. B., Novikova N. O., Zinovatna S. L., Komleva N. O. Automated object-oriented for software module development, *Applied Aspects of Information Technology,* 2021, №4, pp. 338–353.
22. Winkler W. E. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage, *Proceedings of the Section on Survey Research Methods,* 1990, pp. 354–359.

Received 10.09.2022.
Accepted 08.11.2022.

УДК 004.415.2

ВИЗНАЧЕННЯ ВІДНОСИН УСПАДКУВАННЯ ТА РЕСТРУКТУРИЗАЦІЯ МОДЕЛЕЙ ПРОГРАМНИХ КЛАСІВ У ПРОЦЕСІ РОЗРОБКИ ІНФОРМАЦІЙНИХ СИСТЕМ

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

Витнова А. І. – студентка кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», Одеса, Україна.

АНОТАЦІЯ

Актуальність. Реалізація різних варіантів використання може виконуватись різними командами розробників у різний час. Це призводить до створення погано структурованого коду. Проблема ускладнюється при розробці середніх та великих проектів у стислий термін.

Мета. Оскільки успадкування є одним із ефективних способів структурування та покращення якості коду, метою дослідження є визначення можливих зв’язків успадкування для різноманітних моделей класів.

Метод. Запропоновано виділення з множини класів, що представляють модель класів на певному етапі проектування, підмножин, для яких можливий загальний батьківський клас (в окремому випадку абстрактний клас). Для вирішення завдання сформульовано ознаки спільності класів. Удосконалено математичну модель концептуального класу за рахунок включення інформації про обов’язки класу, його методи та атрибути. Встановлено зв’язок кожного класу з сценаріями, для яких він використовується. Запропоновано систему типів даних для елементів моделі класу. Розширено опис сигнатур методів класів. Розроблено метод реструктуризації моделі класів, що передбачає 3 етапи. У першому визначаються коефіцієнти близькості класів. На другому створюються підмножини можливих дочірніх класів. На третьому виконується автоматизоване перетворення структури класів з урахуванням виявлених відносин спадкування.

Результати. Розроблено програмний продукт для проведення експериментів щодо виявлення можливих відносин успадкування залежно від кількості класів та ступеня їхньої подібності. Результати проведених випробувань показали ефективність ухвалених рішень.

Висновки. Метод використовує алгоритм формування підмножин класів, які можуть мати одного предка та алгоритм автоматичного створення та перетворення класів для побудови дворівневої ієрархії класів. Результати дослідження реалізовані у програмному продукті. Експеримент показав триразове скорочення помилок при виявленні наслідування та багаторазове скорочення часу порівняно з існуючою технологією.

КЛЮЧОВІ СЛОВА: модель класу, атрибут класу, метод класу, типи даних, варіант використання, спадкування.

ЛІТЕРАТУРА / ЛІТЕРАТУРА

1. Object-Oriented Analysis and Design with Applications 3rd Edition / [G. Booch, R. A. Maksimchuk, M. W. Engle et al.]. – Boston : Addison-Wesley Professional, 2007 – 694 p.
2. Lee G. Modern Programming: Object Oriented Programming and Best Practices / G. Lee. – Birmingham : Packt, 2019. – 266 p.
3. Baesens B. Beginning Java Programming: The Object-Oriented Approach / B. Baesens, A. Backiel, S. Broucke. – Birmingham : Wrox, 2015. – 672 p.
4. Agility as an innovation driver: towards an agile front end of innovation framework / [M. Brand, V. Tiberius, P. M. Bican, A. Brem]. – Potsdam : Springer, 2021. – P. 157–187.
5. Project Implementation Decision Using Software Development Life Cycle Models: A Comparative Approach / [M. A. Adeagbo, J. Akinsola, A. A. Awoseyi, F. Kasali] // Jour-

DOI 10.15588/1607-3274-2022-4-8

nal of Computer Science and Its Application. – 2021. – № 28. – P. 122–133.
6. Jacobson I. USE-CASE 2.0 The Guide to Succeeding with Use Cases [Electronic Recourse] / I. Jacobson, I. Spence, K. Bittner. – Access mode: https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf
7. Arcos-Medina G. The Influence of the Application of Agile Practices in Software Quality Based on ISO/IEC 25010 Standard / G. Arcos-Medina, D. Mauricio // International Journal of Information Technologies and Systems Approach. – 2020. – №13. – P. 1–27.
8. Mohan M. A survey of search-based refactoring for software maintenance / M. Mohan, D. Greer // Journal of Software Engineering Research and Development. – 2018. – №6. – P. 1–52.
9. Ryan M. Mastering OOP: A Practical Guide to Inheritance, Interfaces, and Abstract Classes [Electronic Recourse] / M. Ryan. – Access mode: https://www.smashingmagazine.com/2019/11/guide-oop-inheritance-interfaces-abstract-classes/
10. Taubler D. When to Use Abstract Classes [Electronic Recourse] / D. Taubler. – Access mode: https://betterprogramming.pub/when-to-use-abstract-classes-70fe526165ac
11. AI-Fedaghi S. Classes in Object-Oriented Modeling (UML): Further Understanding and Abstraction / S. AI-Fedaghi // International Journal of Computer Science and Network Security. – 2021. – №21. – P. 139–150.
12. Minh Hoang Lien Vo Transformation of UML class diagram into OWL Ontology / Minh Hoang Lien Vo, Q. Hoang // Journal of Information and Telecommunication. – 2020. – №4. – Issue 1.
13. Gandhi P. Optimization of Object-Oriented Design using Coupling Metrics / P. Grandhi, K. Pradeep // International Journal of Computer Applications. – 2011. – №27. – P. 41–44.
14. Saeed M. G. Three Levels Quality Analysis Tool for Object Oriented Programming / M. G. Saeed, M. T. Alasaady // International Journal of Advanced Computer Science and Applications. – 2018. – №9. – P. 522–536.
15. Zongmin Ma Two-level clustering of UML class diagrams based on semantics and structure / Zongmin Ma, Zhongchen Yuan, Li Yan // Information and Software Technology. – 2021. – №130. – 106456.
16. Evaluating the effects of similar-class combination on class integration test order generation / [Miao Zhang, Jacky Wai Keung, Yan Xiao, Md Alamgir Kabir] // Information and Software Technology. – 2021. – №129. – 106438.
17. Comparing manual and automated feature location in conceptual models: A Controlled experiment / [F. Pérez, J. Echeverría, R. Lapeña, C. Cetina] // Information and Software Technology. – 2020. – №125. – 106337.
18. Benlhachmi K. A Formal Model of Conformity and Security Testing of Inheritance for Object Oriented Constraint Programming / K. Benlhachmi, M. Benattou // Journal of Information Security. – 2013. – №4. – P. 113–123.
19. Vozovikov Yu. N. Information technology for automated compilation of use cases / Yu. N. Vozovikov, A. B. Kungurtsev, N. A. Novikova // Science practices of Donetsk National Technical University. – 2017. – No. 1 (30). – P. 46–59.
20. Method for defining conceptual classes in the description of use cases / [O. Kungurtsev, N. Novikova, M. Reshetnyak et al.]. – Odessa: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019, 2019 – 1117624.
21. Automated object-oriented for software module development / [O. B. Kungurtsev, N. O. Novikova, S. L. Zinovatna, N. O. Komleva] // Applied Aspects of Information Technology. – 2021. – №4. – P. 338–353.
22. Winkler W. E. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage / W. E. Winkler // Proceedings of the Section on Survey Research Methods. – 1990. – P. 354–359.

DOI 10.15588/1607-3274-2022-4-8