UDK 004.912

Nataliia O. Novikova[1], Senior Teacher of the Department"Technical Cybernetics and Information Technology named Prof. R.V. Merkt", E-mail: nataliya.novikova.31@gmail.com,
ORCID: 0000 – 0002 – 6257 – 9703
[1]Odessa National Maritime University, Mechnikov, str. 34, Odessa, Ukraine, 65029

CHANGING AND TRACING OF SOFTWARE REQUIREMENTS AT LEVEL OF CONCEPTUAL CLASSES

Abstract. *The article explores the problem of automating the description of Use Cases at the stage of forming requirements in the design of software products. Methods for correcting the model of conceptual classes in connection with changes in the formulation of various items of scenarios for Use Cases are proposed and tracing of each item of the Use Case scenario in conceptual classes and their methods and attributes. Changing requirements at the level of use cases description means deleting previously compiled items of the scenario and/or adding new ones. Deleting a Use Case is considered to be a consecutive deletion of all of its items, and editing a scenario item as a deletion followed by a new edition. The methods of removing all types of scenario items of the proposed classification in various possible situations are considered: the class created earlier was not used in other items of this or other use cases; the class created earlier was not used in other items of this or other use cases, but the function contained in the class has references to other functions; the class created earlier was used in other items of this or other use cases and the function contained in the class has no reference to other functions; the class created earlier was used in other items of this or other use cases and the function contained in the class has references to other functions. Methods have been developed for determining the relationships of Use Case and its item with classes, their methods and attributes that implement this item (direct tracing), and determining the relationship of any data element or class method with various Use Case and their items (reverse tracing). The proposed method for conceptual classes correcting allows automatic deleting various items in scenarios while maintaining the correct presentation of conceptual classes. It is shown that there is a significant reduction in time for correcting classes in an automated mode compared to the traditional manual mode. The tracing method also significantly reduces the time it takes to find the connections between the Use Case.*

Keywords: *use cases; scenarios; models; conceptual classes; tracing*

Introduction

Use Cases (UC) is a widespread method for detailed recording of functional requirements for a software product being designed [1-2]. UCs forms the basis of an object-oriented approach to software development [3-4] and is supported by the UML language [5-7]. The whole process of identifying and formulating requirements is long, very responsible, and time-consuming [8-9], and this especially applies to the description of UC [10]. Such work is usually carried out by a system analyst [11], who should not only have in-depth knowledge of information technology but also be a good psychologist and organizer [12-13].

Automation of tasks solved by a system analyst can significantly improve the quality of requirements for a software product; reduce costs and time for their preparation.

Literature analysis

Use Cases are written in the language of the subject area. The terms used in their preparation will be further displayed in user interfaces, in the names of classes and their methods. Therefore, studies on the automation of the compilation of a domain terms glossary deserve attention [14-15]. Also in recent years, studies have appeared on the automation of UC descriptions. In [16], a classification of items of UC scenarios were proposed, on the basis of which tools were developed to automate the compilation of UC descriptions. As a continuation of this line of research, it is proposed to create models of conceptual classes (MCC) in parallel with the UC description. In [17], it was proposed to consider UC in the form of two models – the description model and the design model, which allows the formation of the MCC.

Changing requirements at the first stages of software design is a common occurrence [18,19], so it becomes necessary to display the changes made to the UC description and, in parallel, to the design model. Existing means of displaying project documentation [20-22] use various indexing methods to search for information. In the best case, they allow to visually trace the path from changing a certain requirement to the group of classes that implement it (direct tracing), or the path from changes in some classes to the requirements on the basis of which these classes were created (reverse tracing).

If we are talking about automating the construction of a model of conceptual classes in the process of compiling UC, then this is clearly not enough. Changes introduced into the requirements should be automatically processed and lead to the adjustment of a previously compiled model of conceptual classes, what's one has not been done so far.

Problem statement

Changing requirements at the level of use case description means deleting previously compiled items of the scenario and/or adding new ones. In order to add a new item, it is necessary to qualify it and perform the corresponding algorithm. Algorithms for adding various types of scenario items are described in detail in [16]. The problem is the deletion of an existing item of the scenario since its implementation in the form of conceptual classes can service not only this item of the script, but also other items of the scenario in question, and possibly other scenarios.

To fix this problem, the two tasks should be solved.

1. To develop a method for adjusting the MCC in connection with changes in the wording of various items of UC scenarios.

2. To develop a method for tracing each item in a UC scenario into conceptual classes, their methods and data, as well as a method for tracing any function or given class into the corresponding items in UC scenarios.

Method for conceptual classes adjusting

We will construe the removal of the UC, the removal of the UC item, the change of the UC item as changes of requirements. All these changes come down to deleting one item of the scenario. To do this, we will consider the removal of UC as a sequential deletion of all of its items, and editing a script item as a deletion with subsequent compilation in a new edition. Thus, it is necessary to review the removal of all types of items proposed in [17]:

– **Create**. The user commands the system to create some object which can contain data used both within the framework of this UC and other UC.

– **Enter the data**. The user enters into the system a series of data, for which the system usually must check the possibility of their use for further work.

- Request a value. The user asks the system for some data. This is usually followed by a user's assessment of the data.

– **Request a list**. The user orders a list (for example, data, services or documents) for a further selection of some elements from it.

– **Select from the list**. The user selects the necessary data or service (document) from the list.

– **Enter the service (document)**. The user enters the necessary service or document, which determines the further sequence of actions. For example, a payment method by bank card.

– **Repeat the actions**. The user has the opportunity to go to the above items of the scenario, or refuse to repeat them.

– **Complete the UC**. The item provides for the successful completion of the UC, which may be accompanied by the preservation of certain data, the formation of a report, documents, etc.

We will use the class model proposed in [17]. All classes included in the MCC are represented by the set

$$Mc = \{c\}. \qquad (1)$$

Each class (prototype) is represented by a tuple

$$c = < tc, cName, \tau, uName, nP, mData, mFunc >, \qquad (2)$$

where: $tc = "class" | "prototyp"$;

– $cName$ is the name of the class;

– $uName, nP$ is the name of the UC and the number of the item where the class was created;

– $\tau = "u" | "s"$ is the lifetime of the class objects (during the execution of the use case – u, or during the operation of the system – s);

– $mData$ is the set of attributes that the class contains;

– $mFunc$ is the set of functions (methods) that the class contains.

Elements of a set $mData$ are represented by a tuple

$$data = < dName, td, ref >, \qquad (3)$$

where: $dName$ is the name of the attribute;

$td = "si" | "ar" | "sc" | "ac"$ is the type of attribute (single value, array, calculated single value, calculated array);

$ref = \{< fName, uName, nP >\}$ are references to functions (methods) using the attribute;

where: $fName$ is the name of the method;

$uName, nP$ determine the UC and the item at which data use occurred.

The class method is defined in the following way

$$\begin{gathered} func = < fName, oData, mArgs, mIData, \\ mCData, mRfFunc > \end{gathered}, \qquad (4)$$

where: $oData$ is the value returned by the method; if $oDana = "0"$, then the method returns nothing;

$oData = "b"$, then the method returns a Boolean value, which doesn't have to be saved in the object after the completion of the method. In other cases, the name should be included in the set

$mData$ with the type $td = "sc"|"ac"$ and a reference to this method;

$mArgs$ is the set of arguments of the method;

$mIData$ is the set of class attributes that take on new values;

$mCData$ is the set of class attributes used in the calculations of this method;

$mRfFunc$ is the set of links to external functions (methods of other classes) used in this method.

Each element of the set $mRfFunc$ is represented by a tuple:

– $mRfFunc_i =< cName_j, func_i >$,

where: $cName_j$ is the class to which the external function belongs (in the general case, several external functions may belong to the same class);

$func_i$ is the class $cName_j$ function referenced by the function $func$.

1. Deleting an item of “Create” type

In accordance with [17], the “Create” item provides for the creation of a class in accordance with the following description:

$c_1 =< "class", cName_1, \tau_1, uName_1, nP_1, mData_1, func_1 >$,

where: $cName_1$ is the name of class;

$uName_1, nP_1$ is the name of the use case and the item deleted;

$mData_1$ – class attributes generated after initialization;

$func1 =< fName, "0", mArgs, mIData, mCData, \varnothing >$ is the initialization function of the class object (usually a constructor),

$oData = "0"$ means the method isn't returning a value;

$mArgs = mData^*$;

$mIData = mData_1$;

$mCData$ may differ from $mArgs$, if the default data exists;

$mOFunc = \varnothing$ means when creating an object, external functions are not used.

When deleting an item, two situations are possible.

A. The previously created class was not used at other items in given UC or in other use cases. In this case, it should have only one function – $func1$, i.e.

$mFunc \setminus \{ func1 \} = \varnothing$.

This allows you to remove the class c_1 from a set of classes.

$$Mc := Mc \setminus \{c_1\}. \qquad (5)$$

B. The previously created class was used in other items of the corrected one or other use cases. Then it is necessary to present the class c_1 in the form of a prototype c' and ensure that all items of the scenarios in which this class prototype is used are executed.

Let's define the changes you need to make to the class c_1 to get the class c'.

The class name and the lifetime of the class objects must remain unchanged:

$$cName' = cName_1 \text{ and } \tau' = \tau_1.$$

We define the set of functions that should work with the prototype of the class:

$$mFunc' = mFunc \setminus \{ func1 \}.$$

References in class attributes to a deleted function will be also automatically deleted when a new set of class c' attributes is generated.

To ensure the operation of each function in the class c' the necessary attributes must be stored. We represent the set of functions $mFunc$ in the form:

$$mFunc = \{ func_j \},\ j = 1, n,$$

where: n is the number of functions belonging to this class.

Then the attributes necessary for the operation of all functions from $mFunc$ will be defined as

$$mData' = \bigcup_{j=1}^{n} mCData_j \ . \qquad (6)$$

When creating a class, you must specify the item and Use Case where it was created. When creating a prototype of a class, the item and UC should be indicated where it was in demand. Since the method of constructing conceptual classes does not store the chronology of their creation and modification, the creation of a prototype can be attributed to any UC in which this class is used, and to the first item in which access to c' occur. We write the selection conditions as follows: if

$\exists data_j \in mData' \mid < fName, uName_{j,i}, nP_{j,i} > \in ref_j$,

then we take the $uName' = uName_{j,i}$.

To determine the item number in UC, we form a set of links to functions ordered by increasing item number:

$$mRf \to sort[mRf],$$

then $nP' = nP_1$,

where: nP_1 belongs to the first tuple of mRf.

Finally, the class c' will be presented as

$$c' =< "prototype", cName', \tau', uName'. nP', mData', mFunc' >$$

2. Deleting an item of "Enter the data" type

The "Enter data" item provides for the action with the data of a previously created class (prototype), as well as a possible verification of the accuracy of the data entered.

We denote the UC as $uName1$, scenario item as $nP1$, the set of data entered as $mData_1$, the corresponding function as $func1$, and the class containing the data and the function as c_1.

When deleting an item, the following situations are possible.

A. The class c_1 is not used in other items of this UC or other UCs, and the function $func1$ has no references to other functions. This condition is met if for each data element of the class c_1

$$data = < dName, td, gH, gL, ref >,$$

all function references

$$ref = \{< fName, uName, nP >\}.$$

Contain only names $uName1$ and $nP1$:

$$\forall data_i \in mData1 \mid uName_{i,j} = uName1 \wedge nP_{i,j} = nP1. \quad (7)$$

In this case, we remove the class c_1 from the set of classes

$$Mc := Mc \setminus \{c_1\}$$

B. The class c_1 is used at other items of the corrected UC or other UCs, and the function $func1$ has no references to other functions. In this case, condition (7) is not met and it is necessary to ensure the fulfilment of all items of the scenarios ($nP_i \neq nP1$), in which this class is used.

We define the changes that need to be made to the class c_1 to get the class c', that matches the UC descriptions with the deleted item $nP1$.

The class name and the lifetime of the class objects must remain unchanged:

$$cName' = cName_1 \text{ and } \tau' = \tau_1.$$

The function $func1$, except for the item $nP1$, can be used in other items. Therefore, we perform selective deletion of references.

If

$\exists data_i \in mData1 \mid uName_{i,j} = uName1 \wedge nP_{i,j} = nP1$,

then $ref_{i,j}$ is removed from the set ref_i.

If the condition

$\forall data_i \in mData1 \mid uName_{i,j} \neq uName1 \wedge nP_{i,j} \neq nP1$

Is met, then we delete the function $func1$ from the set of function of class c':

$$mFunc' = mFunc \setminus \{func1\}.$$

If we represent the many functions $mFunc$ of the class c' as

$$mFunc = \{func_j\}\ j = 1, n.$$

Then we can determine the data necessary for the operation of all functions from $mFunc$:

$$mData' = \bigcup_{j=1}^{n} mCData_j.$$

Finally, the class c' will be presented as

$$c' = <"prototype", cName', \tau', uName', nP', mData', mFunc' >.$$

C. The class c_1 was not used in other items of this UC or other UCs, and the function $func1$ has references to other functions. In this case, it is not enough to delete the class c_1. We must also delete the "traces" of calling other functions from the function $func1$.

In accordance with (4), the deleted function has the form

$$func1 = < fName, oData, mData1, mIData, mCData, mRfFunc > .$$

If any function $func_m$ from the set $mRfFunc$ was created in the item $nP1$ UC $uName1$ in a certain class c_j and is not used at other items in the scenarios, then it should be deleted. If the class c_j contains only a function $func_m$, that is used only in the item $nP1$ UC $uName1$, then the class c_j must also be deleted. If $func_m$ used in other items, then only the reference to the item and UC should be deleted.

Let us successively analyze each element of the set of references $mRfFunc$ to functions of other classes.

Consider some element of the set $< cName_j, func_m >$. The fulfilment of the condition

$$\exists\ data_i \in mData_j \mid fName_{i,k} = func_m.fName \wedge uName_{i,j} = uName1 \wedge nP_{i,j} = nP1. \quad (8)$$

Indicates that the function $func_m$ was created in item $nP1$ UC $uName1$. If this condition is not met, then the function $func_m$ from the class c_j is not deleted, but a reference to it is only subjected to this

$$< func_m.fName, uName1, nP1 >\}.$$

If condition (8) is met, then the use of the function $func_m$ by other points in the scenarios should be determined. If the condition

$$\exists\ data_i \in mData_j \mid fName_{i,k} = func_m.fName \wedge uName_{i,j} \neq uName1 \wedge nP_{i,j} \neq nP1 \quad . \qquad (9)$$

Is met, then the function $func_m$ from the class c_j is not deleted, but a reference to it only undergoes this.

If condition (8) is met and condition (9) is not, then the function $func_m$ is deleted from the class c_j

$$mFunc_j := mFunc_j \setminus \{func_m\} .$$

If the class c_j does not contain other functions ($mFunc_j = \varnothing$), then the class is deleted c_j: $Mc := Mc \setminus \{c_j\}$.

Function verification operations are repeated for all elements of the set $mRfFunc$.

Regardless of the results of the analysis $mRfFunc$, the last corrective action is to remove the class c_1:

$$Mc := Mc \setminus \{c_1\} .$$

D. The class c_1 is used at other items of the corrected UC or other UCs, and the function $func1$ has references to other functions. In this case, all operations provided for by options B and C are performed, except for deleting the class c_1.

3. Deleting an item of "Request a value" type

When designing the "Request a value" item, a previously created class or a previously existing class may have been used. In the first case, we need to consider the possibility of deleting this class, in the second – only deleting the corresponding function and data.

We denote the UC considered as $uName1$, the scenario item as $nP1$, the set of requested data values as $mData_1$, the corresponding function as $func1$, and the class containing the data and the function as c_1.

When deleting an item, the following situations are possible.

A. The class c_1 is not used in other items of this UC or other UCs, and the function $func1$ has no references to other functions. In this case, all the actions provided for in subsection 2.A are performed.

B. The class c_1 is used at other items of the corrected UC, or other UCs, and the function $func1$ has no references to other functions. In this case, all the actions provided for in subsection 2.B are performed.

C. The class c_1 is not used in other items of this or other UCs, and the function $func1$ has references to other functions. In this case, all the actions provided for in subsection 2.C are performed.

D. The class c_1 is used at other items of the corrected UC, or other UCs, and the function $func1$ has references to other functions. In this case, all operations provided for by options 2.B and 2.C are performed, except for deleting the class c_1.

4. Deleting an item of "Request a list" type

The process of deleting this item does not differ from that considered earlier in section 3. However, it needs to be borne in mind that the item "Select from the list" should follow the item "Request a list" in the UC scenario. Therefore, in the future, this item should also be deleted.

5. Deleting an item of "Select from the list" type

The scenario item provides for entering a value selected by the user from the list.

The operation of deleting this item is similar to the operation "Enter the data" type. However, it's necessary to note that this item in the UC scenario must be preceded by the "Request a list" item. Therefore, correcting one of these points should be considered as deleting both.

6. Deleting an item of "Enter the service" type

The operation of deleting this item is similar to the operation "Enter the data". However, unlike the "Enter the data" item, the service entered can determine the choice of one or another scenario of working with the system. Therefore, deleting this item requires a detailed analysis of all further items in the main scenario and extension scenarios.

7. Deleting an item of "Repeat the actions" type

The implementation of the "Repeat the actions" scenario item does not add new functions and classes to the kernel of the system since it can be implemented by user interface classes. Therefore, the removal of this item does not lead to the adjustment of the MCC, however, it significantly changes the sequence of execution of the scenario items.

8. Deleting an item of "Complete the UC" type

When creating this item, the following operations may have been performed:

– reception by an existing object of some data $mData_1$, that did not require analysis;

– registration of certain data $mData_2$ in existing facilities;

– creation of a document containing certain data $mData_3$.

Therefore, the process of deleting an item falls into three stages.

8.1 Deleting the data. We denote the UC under consideration as $uName1$, the scenario item as $nP1$, the set of input data as $mData_1$, the corresponding function as $func1$, and the class containing the data and the function as c_1.

Since the class c_1 was not created in $nP1$ item, the question of deleting the class c_1 is not considered. It should be possibly needed to remove a function $func1$ from the set of class c_1 functions and data $mData_1$ from the set of class c_1 data.

If the condition

$$\exists func1 \in mFunc_1 \mid ref_{1,i}.uName \neq uName1 \wedge ref_{1,i}.nP \neq nP1 \quad . (10)$$

Is met then the function $func1$ is not deleted, but only the link to its use in $uName1$ and $nP1$ subjected to this:

$$< fName1, uName1, nP1 > .$$

If condition (10) is not satisfied, then the function is deleted from the class c_1:

$$mFunc1 := mFunc1 \setminus \{func1\},$$

and the class data is adjusted:

$$c_1 : mData1 = \bigcup_{j=1}^{n} mCData_j ,$$

where: $mCData_j$ is determined from the remaining functions

$$mFunc = \{func_j\}, \; j = 1, n .$$

8.2 Cancelling of data registration. When creating this item, the data to be registered was obtained from one object and registered in others. We denote the UC considered as $uName1$, the scenario item as $nP1$, the set of data extracted from the class c_1 object as $mData_1$, the corresponding function as $func1$. Data from the set $mData_1$ can be registered in several objects of different classes. Therefore, when cancelling registration, it may be necessary to delete data from the classes where they were registered; and to delete the corresponding functions of these classes and the function $func1$.

The function that performs registration has the form

$$func1 =< fName_1, "0", \varnothing, \varnothing, mData_1, \{c_j, .func_k\} > . \quad (11)$$

Consider the correction of one of the classes (c_j), in which registration was performed. For registration, a function $fName_k$ of this class was used

$$func_k =< fName_k, "0", mArgs, mIData_1, mCData, \varnothing >,$$

where: $mArgs \in mData_1$ is the part of the registered data.

We analyze the function $func_k$ in the context of its use in other items and UC:

$$\exists func_k \in mFunc_j \mid ref_{j,i}.uName \neq uName1 \wedge ref_{j,i}.nP \neq nP1. \quad (12)$$

If the condition is true, then the function $func_k$ is not deleted. From the set of c_j class references ref only the element $< fName_k, uName1, nP1 >$, representing the reference to the deleted item in the scenario is subjected to deletion.

If condition (12) is not satisfied, then the function $func_k$ must be deleted

$$mFunc_j := mFunc_j \setminus \{func_k\},$$

and the class data must be corrected c_j

$$mData_j = \bigcup_{i=1}^{n} mCData_i ,$$

where: $mCData_i$ is determined from the remaining functions

$$mFunc_j = \{func_i\}, \; i = 1, n .$$

The considered sequence of operations should be performed for each element from the set $\{< c_j, .func_k\}$ from (11).

8.3 Cancelling the document creation. The creation of the document was performed by one class (c_1), that existed previously. The function that creates the document ($func1$) was also used once and did not enter new data into the class. Thus, cancelling the creation of a document is limited to deleting a function $func1$ from class c_1:

$$mFunc_1 := mFunc_1 \setminus \{func1\} /.$$

9. Tracing Method Depending on the task being solved, direct or reverse tracing may be required. By direct tracing, we mean the definition of the connections of UC and its item with classes, their methods and data that implement this item (Fig. 1).

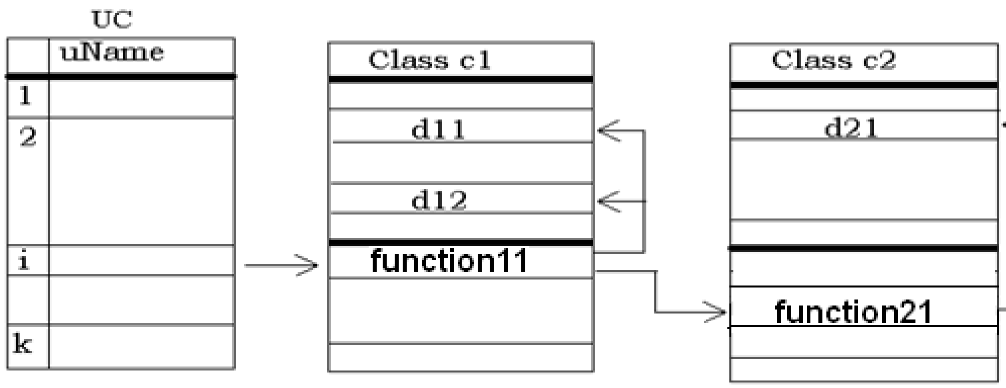

Fig. 1. Tracing from UC and its item to classes, methods and data

By reverse tracing, we mean the definition of the connection for any data element or class method with various UC and their items (Fig. 2).

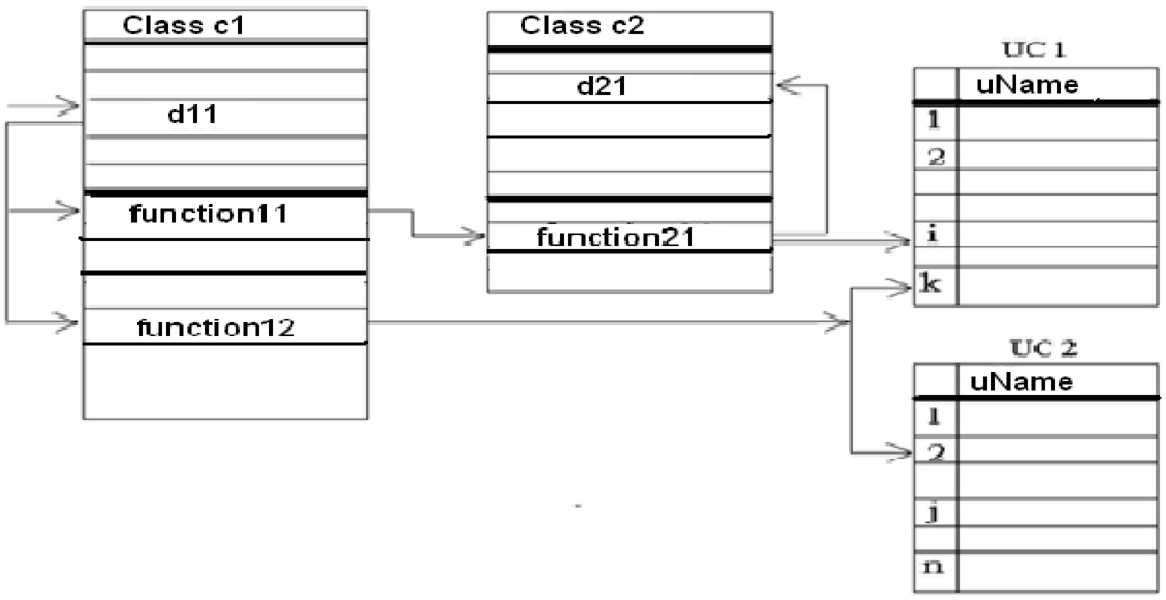

Fig. 2. Tracing from a class method and data to UC and their items

9.1. Direct tracing

We denote the UC under consideration as $uName1$, the scenario item as $nP1$. Let Mct represent a set of classes that are used to implement $nP1$ ($Mct \in Mc$).

Let us represent the result of Trd tracing as a tuple

$$Trd = < UName1, nP1, Mct > .$$

Each class from Mct will be presented as

$$c = < cName, mData, mFunc >$$

We will analyze the data of each class from Mc.

If, for some class $c_i \in Mc$ in the data set $mData_i$ there exists an element

$$\exists data_k \in mData_i \mid ref_{k,j}.uName = Name1 \wedge ref_{k,j}.nP = nP1 ,$$

then the following operations must be performed

A) If

$$\exists c_t \in Mct \mid c_t.cName = c_i.cName, \qquad (13)$$

Then $mData_t := mData_t \cup \{data_k\}$ and $mFunc_t := mFunc_t \cup \{ref_{k,j}.fName\}$.

B) If condition (13) is not met, then a class $c_t = < c_i, \{data_k\}, \{ref_{k,j}.fName\} >$. is created

9.2. Reverse tracing

For reverse tracing, it is necessary to determine the UC and their items from the datum $dName$ belonging to class c_1. Let's represent the result of Trd tracing as a tuple

$$Trr < tData, mUC > ,$$

where: $tData = < c_1, dName > ,$

$-mUC$ is a set of entries for each UC. Each entry has the form

$$< uName, mNP > . \quad (14)$$

Here $uName$ is the name of UC;

mNP is a set of entries of the form $< nP, mFunc >$, where $mFunc$ is the set of functions directly or indirectly using $data$.

We define a set of functions from the class c_1, using $dName$.

Since each datum is represented by a tuple $data =< dName, td, gH, gL, ref >$, then a set of references ref allow us to define all functions using $dName$:

$$ref = \{< fName, uName, nP >\}.$$

We transform each element of the set ref into an element of the set mUC (14):

$$< uName, \{nP, \{c_1 . func\}\} > .$$

Since functions of other classes (not c_1) can use $dName$ through function class c_1, we define these functions by parameter $mRfFunc$ (4).

If for some function of the class c_i the condition

$\exists func_{i,j} \in mFunc_i \mid c_1 . func_k \in mRfFunc_{i,j} >$, is met then we introduce a new element into the set mUC

$$< uName, \{nP, \{c_i . func_j\}\} >,$$

where: $uName$ and nP are given from definition of class c_i in accordance with (2).

After analyzing all classes, it is desirable to combine some elements of the set mUC. If there are two entries, $< uName_i, mNP_i >$ and $< uName_j, mNP_j >$, for which $uName_i = uName_j$, then they are combined into one $< uName_i, mNP_i \cup mNP_j >$.

If there are two entries in the set mNP_i $< nP_i, mFunc_i >$ and $< nP_j, mFunc_j >$ for which $< nP_i = nP_j >$ then they are combined into one $< nP_i, mFunc_i \cup mFunc_j >$.

Testing the results of the study

By testing we mean two types of work:

– checking the correctness of changes in the structure of classes when editing various items of the scenario;

– assessment of the reduction of time for changing the structure of classes in the conditions of application of the proposed method of automation of class correcting.

For the first study, the following Use Case was described in Photo_studio system.

Title: “Accepting orders in the photo studio”.

Level: UC of the user's goal level.

Main actor: order taker (T).

Interested party: customer (C).

Software product: photo studio automation system (S).

Table 1 shows the main successful UC scenario, as well as the classes and methods that were created for each item in the scenario.

result of Trd tracing as a tuple

Table 1. UC scenario, classes and methods for implementing scenario items

Use Case		Classes used	Methods used
Item No.	Item content		
1	C asks T for the provision of the service. T creates a new order in S. (Create)	SOrder	create()
2	C informs about the type of service. T enters data in S. S confirms it. (Enter the data)	SOrder, ServList	ServList.isService(ser), SOrder.setServ(ser)
3	C informs about format. T enters data in S. S confirms it. (Enter the data)	SOrder, ServList	ServList.isFormat(form), SOrder.setFormat(form)
4	C informs about the number of copies. T enters data in S. S confirms it. (Enter the data)	SOrder, ServList	ServList.isCopies(n), SOrder.setCopies(n)
5	C informs about the desired lead time. T enters data in S. S confirms it. (Enter the data)	SOrder, ServList, OrderList	ServList.getDeadline(ser, Form,n), OrderList.getDeadline(),SOrder.s etDeadline(dl)
6	T asks S for the cost of the work. S reports it. C agrees. (Request a value)	SOrder, Tariff	Tariff.getCost(ser,form,n), SOrder.setCost(cost)
7	C contributes a certain amount of money. T enters data in S. S counts the back-giving change. (Enter the data)	SOrder	SOrder.getChange (amount)
8	C informs about the full name and contacts. P captures the received data in S. S passes the entry transaction in the Journal and the order in the list of orders, generates a receipt. (Data Registration)	SOrder, OrderList, Register	SOrder.setName(sname, Phon),OrderList.addOrder(sorder) , Register.addOrder(sorder), SOrder.printReceipt()

Table 2 shows the changes in the requirements (replacement, correcting, adding new items to the scenario). The analysis of changes in items of the scenario of four types is presented. The intermediate state of the class structure (deleting an item in the old edition) and the final state were recorded.

Table 2. Assessment of the correctness of changes in the structure of classes as a result of changes in requirements

No	Scenario item new edition	Item type	Class Structure Changes		Consistency with expected results
			After scenario item deleting	For the new edition of item	
1	C asks T for the provision of the service. T creates in S a new order with the full name of C	Create	SOrder class is deleted (item 1). SOrder prototype is created (item 2). All references are switched to the prototype	The prototype SOrder is switched into class status. The create(fio) method has been added. All references switched to the class	Yes
2	Instead of item 2. 2. T asks S for a list of services. C outputs the data. 3. C chooses a service. T captures the service data in S.	Output the list Enter the data	The ServList class is deleted. ServList.isService(ser) and SOrder.setServ(ser) methods are deleted. The ServList prototype is created in item 3	The ServList.getList() method is added in class prototype ServList The SOrder.setServ(ser) method is added	Yes
3	C informs about full name and contacts. T captures the received data in S. S passes the entry transaction in the Journal and the order in the list of orders, indicating the full name and contacts of C, generates a receipt	Data logging	SOrder.setName (sname,Phon), OrderList.addOrder (sorder), Register.addOrder (sorder), SOrder.printReceipt() methods are deleted	SOrder.setName (sname, Phon), OrderList.addOrder (sorder), Register.addOrder (sorder), SOrder.printReceipt () methods are restored. (The full name of C entered as an attribute in item 1 of the scenario)	Yes

Similar experiments have been performed with items of scenarios of other types. In all cases, the results obtained were in line with the expected ones.

For the second study, a group of students of 10 people was involved. In the Photo_studio system, 6 UCs were sequentially described for one project. Then changes have been made to the items in the scenario. The time was determined during which each student in the traditional way (manually) will correct the class system independently for one, two... six precedents. The complexity of the class structure was determined by the number of classes included in it. The obtained dependence is presented in Fig. 3. It is obvious that with an increase in the number of classes, the time for their correcting significantly increases; whilst the execution of this procedure in an automated mode does not exceed fractions of a second.

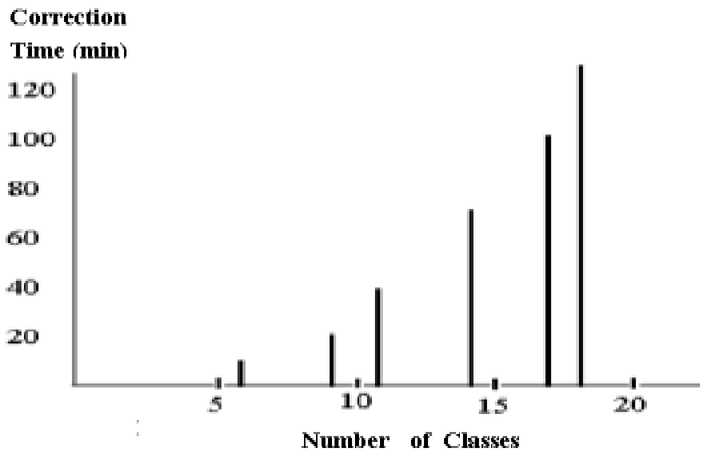

Fig. 3. The dependence of the classes structure correcting time vs the number of classes

Conclusions. The article discusses the problem of automating the description of Use Case at the stages of formation and clarification of functional requirements for the designed software product. When requirements are changed at the first stages of software product design, it becomes necessary to display the changes made to the UC description, and in parallel to the design model. Correcting scenario items in the traditional way usually requires more time.

To solve such problems, an algorithm has been developed for the automated correcting of the conceptual class model in connection with the removal of existing items of the UC scenario of various types. A method for tracing each item in a UC scenario to conceptual classes, their methods and data, as well as a method for tracing any function or this class to the corresponding items in UC scenarios is also proposed.

During the experiments, it was shown that changing items of the scenario of various types lead to adequate changes in the structure of classes (models of conceptual classes). And also the experiments showed the effectiveness of the proposed methods from the point of view of a significant reduction in time for adjusting classes in an automated mode compared to the traditional manual mode.

The proposed method can be used in various technologies of object-oriented design based on the use of UC, at the stage of constructing models of conceptual classes and specifications of program classes.

References

1. Kobern, Alister. (2002). “Sovremennye metody opisaniya funkcional'nyh trebovanij k sistemam”. [Modern methods for describing functional requirements for systems]. Moscow, Russian Federation, *Publ. Lori*, 266 p. (in Russian).

2. Frank, Armour & Miller, Granville. (2000). “Advanced Use Case Modeling: Software Systems”, *Publ. Addison-Wesley*, 425 p.

3. Leffingwell, Dean & Widrig, Don. (Dec 7. 2012). “Managing Software Requirements: A Use Case Approach, Addison-Wesley Professional”.

4. Alexander, Ian & Maiden, Neil. (2004). “Scenarios, Stories, Use Cases”, *Pybl. Wiley*.

5. Wazlawick, Raul S. (2014). “Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL, and IFML”. Morgan Kaufmann, 376 p.

6. Bittner, Kurt & Spence, Ian. (Aug. 20.2002). “Use Case Modeling”. *Addison-Wesley Professional*, 368 p.

7. Dobing, B. & Parsons, J. (2000). “Understanding the Role of Use Cases in UML: A Review and Research Agenda”. *Journal of Database Management*, Vol. 11, No. 4, pp. 28-36. Doi: 10.4018/978-1-931777-12-4.ch008.

8. Vigers, Karl & Bitti, Dzhoj (2014). “Razrabotka trebovanij k programmnomu obespecheniyu”. [Software requirements development], *Publ. BHV*, 736 p. (in Russian).

9. Davis, Alan Mark. (2005). “Just Enough Requirements Management: Where Software Development Meets Marketing”. *Dorset House*, 240 p.

10. Irwin, G. & Turk, D. (2005). “An Ontological Analysis of Use Case Modeling Grammar”. *Journal of the Association for In Formation Systems*, Vol. 6, No. 1, pp. 1-37. DOI: 10.17705/1jais.00063.

11. Matsuura, S. Ogataand. (2013). “A review method for UML Requirements analysis model employing system-side prototyping”. *Springerplus*, Vol. 2, No. 1, 134 p. Doi: 10.1186/2193-1801-2-134

12. Leffinguell, D. Uidrig. (2002). “Principy raboty s trebovaniyami. Unificirovannyj podhod”. [Principles of working with requirements. Unified approach]. Moscow, Russian Federation, *Publ. Izdatel'skij dom “Vil'yams”*, 450 p. (in Russian).

13. Kohn, Mike. (2019). “Pol'zovatel'skie istorii: gibkaya razrabotka programmnogo obespecheniya”. [User stories: agile software development] (Signature Series), *Publ. Dialektika-Vil'yams*, 256 p. (in Russian).

14. Kungurtsev, A. B., Potochnyak, I. V. & Siliaev, D. A. (2015). “Metod avtomatizirovannogo postroeniya tolkovogo slovarya predmetnoj oblasti” [Method for automated construction of a subject dictionary]. *Tekhnologicheskij audit i rezervy proizvodstva*, No. 2/2(22), pp. 58-63 (in Russian).

15. Kungurtsev, O., Zinovatnaya, S., Potochniak, Ia. & Kutasevych, M. (2018). “Development of information technology of term extraction from documents in natural language”. *Eastern-European Journal of Enterprise Technologies*, Vol 6, No. 2 (96), pp. 44-51. DOI: https://doi.org/10.15587/1729-4061.2018.147978.

16. Vozovikov, Yu. N., Kungurtsev, A. B. & Novikova, N. A. (2017). “Informacionnaya tekhnologiya avtomatizirovannogo sostavleniya variantov ispol'zovaniya”. [Information technology for automated use cases]. *Naukovi praci Donec'kogo nacional'nogo tekhnichnogo universitetu*. Pokrovs'k, Ukraine, No. 1(30), pp. 46-59 (in Russian).

17. Kungurtsev, O., Novikova, N., Reshetnyak, M., Cherepinina, Ya., Gromaszek, K. & Jarykbassov, D. (6 November 2019). “Method for defining conceptual classes in the description of use cases”. Proc. SPIE 11176, Photonics Applications in Astronomy, Communications, Industry and High-Energy Physics Experiments, 1117624. DOI: 10.1117/12.2537070.

18. Gottesdiener, Ellen. (2005). “The Software Requirements Memory Jogger: A Desktop Guide to Help Business and Technical Teams Develop and Manage Requirements”. *Addison-Wesley*, 360 p.

19. Hall, E.; Jackson, K. & Dik, D. (2005). “Razrabotka i upravlenie trebovaniyami”. [Development and requirements management]. *Telelogic*, 226 p. (in Russian).

20. “Informacionnye tekhnologii upravleniya. Metody poiska tekstovoj informacii”. [Information Technology Management. Text Information Search Methods]. [Electronic resource]. – Access mode: URL https://refdb.ru/look/2575304-p10.html. – Active link – 02.12.2007 (in Russian).

21. “Indeksy. Teoreticheskie osnovy” [Indices.Theoretical basis]. [Electronic resource]. – Access mode: URL http://www.sql.ru/articles/mssql/03013101indexes.shtml – Active link – 05.10.2003 (in Russian).

22. Ratcliffe, Martyn & Budgen, David. (2005). “The application of use cases in systems analysis and design specification”. *Information and Software Technology*, Volume 47, Issue 9 pp. 623-641. DOI: 10.1016/j.infsof.2004.11.00.

Received 30.01.2020
Received after revision 15.02.2020
Accepted 18.02.2020

УДК 004.912

[1]**Новікова, Наталія Олексіївна**, ст. викладач каф. «Технічна кібернетика та інформаційні технології ім. проф. Р.В. Меркта», E-mail: nataliya.novikova.31@gmail.com,
ORCID: http:// orcid.org/0000 – 0002 – 6257 – 9703
[1] Одеський національний морський університет, вул. Мечникова, 34, м. Одеса, Україна, 65029

ЗМІНА І ТРАСУВАННЯ ВИМОГ ДО ПРОГРАМНОГО ПРОДУКТУ НА РІВНІ КОНЦЕПТУАЛЬНИХ КЛАСІВ

Анотація. *У статті досліджується проблема автоматизації опису варіантів використання на етапі формування вимог при проектуванні програмних продуктів. Запропоновано методи коригування моделі концептуальних класів у зв'язку зі змінами в формулюванні різних пунктів сценаріїв варіантів використання або Use Case і трасування кожного пункту сценарію Use Case в концептуальні класи, в їх методи і атрибути. Зміна вимог на рівні опису прецедентів означає видалення раніше складених пунктів сценарію і / або додавання нових. Видалення Use Case розглядається як послідовне видалення всіх його пунктів, а редагування пункту сценарію - як видалення з подальшим складанням у новій редакції. Розглянуто способи видалення всіх типів пунктів сценарію запропонованої класифікації в різних можливих ситуаціях: створений раніше клас не використовувався в інших пунктах даного, або інших прецедентів; створений раніше клас не використовувався в інших пунктах даного, або інших прецедентів, але функція, що міститься в класі, має посилання на інші функції; створений раніше клас використовувався в інших пунктах даного, або інших прецедентів і функція, що міститься в класі, не має посилання на інші функції; створений раніше клас використовувався в інших пунктах даного, або інших прецедентів і функція, що міститься в класі, має посилання на інші функції. Розроблено методи визначення зв'язків Use Case і його пункту з класами, їх методами і атрибутами, які реалізують цей пункт (пряме трасування) і визначення зв'язку будь-якого даного або методу класу з різними Use Case і їх пунктами (зворотнє трасування). Запропонований метод коригування концептуальних класів дозволяє в автоматизованому режимі видаляти різні пункти сценаріїв, зберігаючи коректне уявлення концептуальних класів. Показано, що спостерігається істотне скорочення часу на коригування класів в автоматизованому режимі порівняно з традиційним ручним режимом. Метод трасування також істотно скорочує час на пошук зв'язків між Use Case.*

Ключові слова*: варіанти використання; сценарії; моделі; концептуальні класи; трасування*

УДК 004.912

[1]**Новикова, Наталия Алексеевна,** старший преподаватель кафедры «Техническая кибернетика и информационные технологии им. проф. Р. В. Меркта», E-mail: nataliya.novikova.31@gm;il.com, ORCID: 0000 – 0002 – 6257 – 9703
[1] Одесский национальный морской университет ул. Мечникова, 34. г. Одесса, Украина, 65029

ИЗМЕНЕНИЕ И ТРАССИРОВАНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ПРОДУКТУ НА УРОВНЕ КОНЦЕПТУАЛЬНЫХ КЛАССОВ

Аннотация. *В статье исследуется проблема автоматизации описания вариантов использования на этапе формирования требований при проектировании программных продуктов. Предложены методы корректировки модели концептуальных классов в связи с изменениями в формулировке различных пунктов сценариев вариантов использования или Use Case и трассировки каждого пункта сценария Use Case в концептуальные классы, в их методы и атрибуты. Изменение требований на уровне описания прецедентов означает удаление ранее составленных пунктов сценария и/или добавления новых. Удаление Use Case рассматривается как последовательное удаление всех его пунктов, а редактирование пункта сценария – как удаление с последующим составлением в новой редакции. Рассмотрены способы удаления всех типов пунктов сценария предложенной классификации в различных возможных ситуациях: созданный ранее класс не использовался в других пунктах данного, либо иных прецедентов; созданный ранее класс не использовался в других пунктах данного, либо иных прецедентов, но функция, содержащаяся в классе, имеет ссылки на другие функции; созданный ранее класс использовался в других пунктах данного, либо иных прецедентов и функция, содержащаяся в классе, не имеет ссылки на другие функции; созданный ранее класс использовался в других пунктах данного, либо иных прецедентов и функция, содержащаяся в классе, имеет ссылки на другие функции. Разработаны методы определения связей Use Case и его пункта с классами, их методами и атрибутами, реализующими этот пункт (прямая трассировка) и определения связи любого данного или метода класса с различными Use Case и их пунктами (обратная трассировка). Предложенный метод корректировки концептуальных классов позволяет в автоматизированном режиме удалять различные пункты сценариев, сохраняя корректное представление концептуальных классов. Показано, что наблюдается существенное сокращение времени на корректировку классов в автоматизированном режиме сравнительно с традиционным ручным режимом. Метод трассировки также существенно сокращает время на поиск связей между Use Case.*

Ключевые слова: *варианты использования; сценарии; модели; концептуальные классы; трассировка*

Nataliia O. Novikova, Senior teacher
Research field: Automation of Information Systems Design