COMPUTER AND INFORMATION NETWORKS AND SYSTEMS. MANUFACTURING AUTOMATION

КОМП'ЮТЕРНІ Й ІНФОРМАЦІЙНІ МЕРЕЖІ І СИСТЕМИ. АВТОМАТИЗАЦІЯ ВИРОБНИЦТВА

UDC 004.912

O.B. Kungurtsev[1], PhD, Prof.,
Nguyen Tran Quoc Vinh[2], PhD,
N.O. Novikova[3]
[1] Odessa National Polytechnic University, 1 Shevchenko Ave., Odessa, Ukraine, 65044; e-mail: abkun@te.net.ua
[2] The University of Da Nang – University of Education, 459 Ton Duc Thang, Da Nang, Vietnam, e-mail: ntquocvinh@ued.udn.vn
[3] Odessa National Maritime University, 34 Mechnykova Str., Odessa, Ukraine, 65029

TECHNOLOGY FOR TESTING OF SOFTWARE MODULES BASED ON USE CASES

О.Б. Кунгурцев, Нгуэн Чан Куок Винь, Н.О. Новікова. **Технологія тестування програмних модулів на основі варіантів використання.** Розроблено автоматизовану технологію, що поєднує процеси опису варіанта використання (прецеденту) і складання наборів тест-кейсів. Для цього запропонована математична модель прецеденту, що представляє його у вигляді орієнтованого графа. Кожна вершина графа відповідає пункту прецеденту, а кожна дуга визначає умови переходу і дані, що визначають ці умови. Застосована класифікація пунктів сценаріїв прецеденту, що дозволила виділити 7 типів пунктів. Для кожного типу пункту прецеденту розроблен окремий шаблон тест-кейса. Шаблони мають 3 розділа: дані, що вводяться в даному пункті, дані що раніше надійшли в систему, та результати виконання пункту. Розроблен алгоритм виявлення незалежних шляхів і процедура визначення завершення обходу. Розроблені програмні засоби, що підтримують запропоновану технологію складання тест-кейсів. Проведені випробування показали істотне скорочення часу при використанні даної технології в порівнянні з існуючими рішеннями, які передбачають роздільні процеси опису прецедентів і складання тест-кейсів.
Ключові слова: варіанти використання, математична модель, тестування, шаблон тест-кейса

O.B. Kungurtsev, Nguyen Tran Quoc Vinh, N.O. Novikova. **Technology for testing of software modules based on use cases**. An automated technology is developed that combines the processes of describing of the use case (precedent) and compiling of test cases sets. For this purpose, a mathematical model of the precedent is proposed, representing it in the form of an oriented graph. Each vertex of the graph corresponds to a precedent item, and each edge defines the transition conditions and data that define these conditions. The classification of the test case scenarios was used, which made it possible to distinguish 7 types of items. For each type of test case item, a separate test case template has been developed. Templates have 3 sections: data entered in this item; data previously received in the system; and the results of the implementation of the item. An algorithm for identifying independent paths and a procedure for determining the completion of traversal is developed. Software tools that support the proposed technology of drawing test cases have been developed. The tests showed a significant reduction in the time when this technology was used in comparison with existing solutions which provide the separate processes for describing of use cases and drawing up of test cases.
Keywords: information technology, use cases, mathematical model, test case, directed graph

Introduction. Presendents (use cases) are often used as a means of defining functional requirements for a software product (SP) being designed [1, 2]. In work [3], based on the study of actions performed by use cases, a classification of possible items of scenarios was created. There the models for the automated compilation of texts of each item type were proposed and the data structures supporting the execution of scenarios were defined.

There are many testing techniques based on the use case description [4, 5]. They assume that the process of testing of the software module should develop according to the scenarios of the relevant use case. However, they do not include algorithms for traversing of all scenarios as well as the procedure for preparing initial data and generating of the expected results.

DOI: 10.15276/opu.3.53.2017.11

ISSN 2076-2429 (print)
ISSN 2223-3814 (online)

Description of the precedent (use case) is made by the system analyst with the direct participation of the representative of the customer. In this process, data entered into the system, data received from the system, as well as their relations, at which certain items of the scenario are executed (or not performed) are to be considered. Usually this information is required only to understand the essence of the problem being solved and is not used after the description of the use case.

However, since the description of the use case is the source material for the programmer, he will soon need information about the data to write the code and certain data sets for testing of the software modules, subsystems and the entire software product. The ability to form test cases at the requirements analysis stage will significantly reduce the total time for the development of the software product. It also opens up the possibility of a partial or full application of the technology "development through testing" (test-driven development, TDD) [6]. In work [7] the technology of TDD application is considered in detail, however there are no solutions for constructing tests based on a formalized description of use cases. In work [8] emphasis is placed on supporting the connection of testing and working with requirements. The following structure of the test case is proposed:

– Sequence number of the step;
– Impact on the system;
– Reference to data;
– Expected Result.

However, the author does not link the methods of presenting requirements with the formation of a test case. An analysis of many factors affecting the quality of unit testing was considered in [9], where as a module one class is adopted, or a set of several classes. Combining several classes that implement the functions of one use case into one module is not analyzed in the work.

The purpose of this work is to reduce the time and improve the quality of unit testing by automating of the preparation of test case sets at the stage of writing and agreeing on a precedent. For this, the necessary tasks are formulated:

– development of a mathematical model of use case;
– determination of paths to bypass the branches (scenarios) of the use case;
– development of templates for test cases corresponding to the classification of scenario items [3].

Example of use case description

Automating the preparation of the use case testing requires a certain formalization of its description. This work was carried out in the research [3], where algorithms and a software solution for the automated description of the use case are proposed. As an example, to which we will refer in the future, we will give a simplified description of the precedent for the sale of a train ticket.

Use case for the sale of a train ticket

Actors: cashier.

Scope: ticket sales subsystem.

Preconditions: the cashier is identified.

Postcondition: Sales data saved. Taxes are correctly calculated. The accounting data has been updated. Commission fees are accrued. The ticket is generated. Payment authorization completed.

Triggers: client's request to the cashier.

The main successful scenario:

1. The client addresses to the cashier with the purpose to get the ticket. The cashier opens a new sale.

2. The client informs the station of departure and destination. The cashier enters the received data into the system. The system confirms the existence of the route.

3. The client reports the departure date, class of the train. The cashier enters the data received from the passenger. The system determines the number of the train and informs the cashier about the availability of the train on the specified date.

4. The client reports the departure date and class of the place. The cashier enters the data received from the passenger. The system determines the number of the car, seats and informs the cashier about the availability of space.

5. The cashier asks for the fare. The system determines the cost of the ticket and tells it to the cashier. The ticket price is calculated using a set of rules. The cashier informs the passenger of the ticket price. The client agrees.

6. The cashier proposes to pay the ticket. The client transfers the amount for payment. The cashier fixes the amount deposited. The system registers ticket sales. Calculates change and generates and issues a ticket. The cashier gives the client a ticket and change.

Extensions (alternative scenarios):

2.a No departure station found.

2a.1. The cashier offers the client to change the name of the departure station. Go to step 2.

2.b No arrival station found.

2b.1. The cashier offers the client to change the name of the arrival station. Go to step 2.

2c There is no message between the specified stations.

2c.1. Completion of the use case.

3a Missing train on the specified date.

3a.1 The cashier offers the client to change the departure date. The client agrees. Go to step 3.

3a.1a Customer disagrees.

3a.1a.1. The cashier offers the client to indicate another class of the train. The client agrees. Go to step 3.

3a.1a.1.a Client does not agree.

3a.1a.1.a.1 Completion of the use case.

3b There is no train of the specified class.

3b.1. The cashier offers the client to indicate another class of the train. The client agrees. Go to step 3.

. . . . .

When describing the use case, the following rules for the numbering of its points are adopted. The points of the main scenario are numbered by decimal numbers. The recommended number of points is not more than 10 [2]. The transition condition number in the alternative scenario begins with the number of the corresponding point of the original script, to which the Latin letter is added. The numbers of the items of the alternative scenario are formed from the condition number, followed by a period and a decimal number.

The use case model

Imagine an use case under consideration in the form of an directed graph [10] without loops and multiple edges.

$$G = (S, U), \tag{1}$$

where S – set of the vertices of the graph,

U – set of the edges.

A graphical representation of a fragment of the use case is shown in Fig. 1. Here single arrows denote edgs leading to the vertices of the graph, and double edges leading to the imaginary top of completion of the use case.Each edgs is represented by a tuple of the form:

$$u_{i,j} = \{< s_i, s_j, d_i, \text{cond}(d_i) >\}, \tag{2}$$

where s_i – nonincident vertex, a s_j – incidental;

d_i – data associated with the vertex s_i;

cond (d_i) – the condition that determines the transition along the edge from the vertex s_i to the vertex s_j. The condition can be expressed by some function that defines a Boolean value *true* or false, or a constant of true if there is an unconditional jump. If the connection between the vertices s_i and s_j is not determined by some condition, but exists permanently, then instead of $\text{func}(d_i)$, it should be written true.

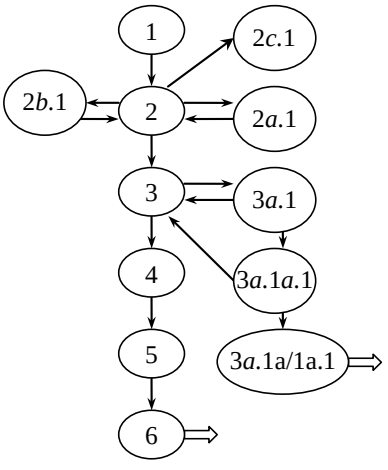

Fig. 1. Representation of an use case in the form of a graph

ISSN 2076-2429 (print)
ISSN 2223-3814 (online)

If the connection between the vertices s_i and s_j is never formed, then instead of $func(d_i)$, we should write false.

To work with a graph, it is usually represented as an adjacency matrix or an adjacency list. The following is a fragment of the adjacency matrix for the use of the "train ticket sale" use case. (Table 1). Here, the node named "7" indicates the external space (completion of the use case). Boolean values or transition conditions are written in the cells of the matrix. For example, if there is no departure station *St*1, then the transition condition from vertex 2 to vertex 2a.1 will be fulfilled – the function is *St*(*St*1) will not find station *St*1.

Table 1

Adjacency matrix

Nodes	1	2	3	2a.1	2b.1	2c.1	7
1	*X*	true	false	false	false	false	false
2	false	*X*	$if\ (isSt(St1) \wedge isSt(St2) \wedge isItinerary(St1, St2))$	$isSt(St1) = = false$	$isSt(St2) = = false$	$isItinerary(St1, St2) = = false$	false
3	false	false	*X*	false	false	false	false
2a.1	false	true	false	*X*	false	false	false
2b.1	false	true	false	false	*X*	false	false
2c.1	false	false	false	false	false	*X*	*true*

Since the adjacency matrix takes up a lot of space, we will use adjacency lists in the future. An adjacency list is assigned to each vertex and specifies the vertices to which to jump from this vertex.

Since the logic of executing and testing the use case provides for the location of the traversal paths of the graph in a certain sequence, we introduce the concept of the level of the vertices of the graph.

To the first level we assign the vertices (1, 2, 3,...) that are included in the main successful scenario (the names do not contain the symbol ".").

To the second level, we assign the vertices (2a.1, 2b.1, 2c.1, 3a.1) of the additional scenario (the names contain one "." Symbol).

The third level includes the vertices of the supplementary scenario from the supplementary scenario (3a.1a.1). These names contain two "." Symbols, etc.

For some *NameS* vertex, we represent the *ListNameS* adjacency list in the form of a set ordered by vertex names:

$$ListNameS, \{NameS_i, LevS_i, TypeS_i, Cond_i, Lab_i\}, \quad (3)$$

where $NameS_i$ – any vertex attainable from *NameS*;

$LevS_i$ – level of the vertex;

$TypeS_i$ – type of the vertex (1 – root, 0 – ordinary);

$Cond_i$ – jump condition to the vertex $NameS_i$;

Lab_i – a note on the possibility of crossing the edge. Possible values for Lab_i: 0 – open, 1 – closed.

Determining independent paths

Use case-based testing involves the preparation of such input data that will provide access to all independent paths of the graph. Known algorithms for finding independent paths in the flow graph [10]. However, when constructing the graph of a use case, predicate nodes can not be formed that represent only simple conditions. The style of writing a use case provides that a step of the scenario can correspond to an unlimited number of steps in the expansion scenario. This leads to the need to develop new algorithms for traversing the graph. For the same reason, it is possible to use only one formula for determining cyclomatic complexity *V*(*G*), which is based on counting all edges and vertices:

$$V(G) = Nu - Ns + 2, \quad (4)$$

where *Nu*– Number of edges;

Ns– Number of vertices.

For the use case fragment shown in Figigure. 1, number of independent paths will be 5.

Traversing strategy

1. For all paths, the initial vertex is the vertex with the name "1".
2. Each path ends with one of the end vertices.
3. Each path traveled is remembered and does not repeat.
4. The first path is the main successful scenario (first level vertices).
5. Each new path must differ from the previous one by the minimum number of new edges and vertices.

Algorithm for determining independent paths of use case graph

Denote the name of the current vertex – *NameScur*.

Denote the name of the next vertex – *NameSnext*.

We will maintain independent paths in the list of independent paths (LIP).

Define the procedure for completing the path when the final vertex is reached.

Procedure for completing a traversing finish (*NameSnext*)

If the *NameSNext* is end vertex with level $L>1$, then we set $Lab_i=1$, we fix the traversed path in LIP. Go to to the Step 2 of the algorithm.

Below are the steps of the algorithm.

1. Pass all the vertices of level 1 from initial to ending. Form the path. Fix the path to LIP number 1.
2. If the number of paths in LIP = $V(G)$, then complete the algorithm. Otherwise, start a new traveling from the initial vertex.
3. Determine the nearest *NameScur* vertex that has the path to the vertex of the next level. Select the vertex of the next level *NameSnext* with the smallest number and the open edge ($Lab_i=0$).
4. Call the procedure finish (*NameSnext*). Go to the vertex *NameSnext* (*NameScur = NameSnext*).
5. If the *NameScur* vertex has an open edge to the vertex of the next level *NameSnext*, go to step 4. Otherwise go to step 6.
6. If the vertex has an reachable edge to the vertext of its *NameSnext* level, then call the procedure finish (*NameSnext*) and go to step 5. Otherwise go to step 7.
7. If the vertex has an available edge to the vertex of the lowest level of *NameSnext*, then mark the edge to the vertex of *NameSnext* as closed and go to the *NameSnext* t node. Go to item 8. Otherwise, mark the traversed edge to the vertex of *NameScur* as closed and go to step 2.
8. If the level of current node is of 1, then complete the traversal of the nodes of the first level. Fix the path to LIP, go to step 2. Otherwise, go to step 6.

As an example in Table 2 shows the LIP obtained for the graph in Fig. 1.

Table 2

Independent paths for the use case fragment "Ticket Sale"

Path number	Path	Comment
1	$1\to2\to3\to4\to5\to6$	–
2	$1\to2\to2a.1\to2\to3\to4\to5\to6$	–
2	$1\to2\to2a.1$	Dead end
3	$1\to2\to2b.1\to2\to3\to4\to5\to6$	–
3	$1\to2\to2b.1$	Dead end
4	$1\to2\to2c.1$	–
5	$1\to2\to3\to3a.1\to3a.1a.1\to3a.1a.1a.1$	–
6	$1\to2\to3\to3a.1\to3a.1a.1\to3\to4\to5\to6$	–
7	$1\to2\to3\to3a.1\to3\to4\to5\to6$	–

Development of test case templates

The set of test variants consists of the groups of tests necessary for passage along each independent path. The presentation of the input data and the expected results depends on the actions envisaged

ISSN 2076-2429 (print)
ISSN 2223-3814 (online)

in each step of the scenario. In [11], a classification of the scenario steps is proposed. 8 types of items were considered totally.

– Create.
– Enter data.
– Request value.
– Request a list.
– Choose from the list.
– Enter the service (document).
– Repetition of actions.
– User action that does not fit into the proposed classification.

Testing the step "Create"

This item can have the condition of creating a textCondition object and a list of parameters for the object being created. Consider the possible options for preparing data and transitions to other steps in the scenario.

A) There are no parameters, there is no condition for creating the object. As a result of the execution of the item, an object is created that has only a name and has no data, or has data set by default. The test version contains only the name of the object.

The fact of creating an object is not directly checked. Because a new object is created for storing and working with data, its testing will obviously be performed when operations with this object are performed. The step provides an unconditional jump to some node.

B) There is no condition for creating an object, there are parameters, there are no parameters checking. The test version contains the name of the object and the parameters.The item provides an unconditional transition to some node.

V) There is a condition for creating an object, there are no parameters. The test version contains the name of the object and some characteristic (for example, the term of existence).

The step provides for the jump to the node *S*1 or to the node *S*2.

G) There is a condition for creating an object, there are unverifiable parameters. The test version contains the name of the object, a list of parameters and some characteristic of the object.

The step provides for the jump to the node *S*1 or to the node *S*2.

D) There is no condition for creating the object, there are checked parameters. The test version contains the name of the object and a list of parameters.

The step provides for the transition to the node S1 or to the nodes *S*2, *S*3, ..., *S*n.

E) There is a condition for creating an object, there are checked parameters. The test version contains the name of the object, a list of parameters and some characteristics. Some other object in the system has conditions for receiving parameters and a valid and invalid characteristic value. Table 3 presents a template for testing the step "Create" for the most general case. If for a specific case some input data are missing, then the corresponding positions of the template are not filled.

Testing the item "Enter data"

This step provides for data entry. Some data may require verification (Table 4).

If for some input data there is no verification, then the receiving condition is represented by a constant of true.

The step provides a jump to the node *S*1 or to the nodes *S*2, *S*3, ..., *Sn* (the conditions for data input are not fulfilled).

Testing the item "Request value"

This item provides for the receipt from the system of the value of some variable, possibly, if certain conditions are met (Table 5). The user retains the right to accept the received value or reject it.

The consent or disagreement of the user with the received value is realized at the level of the system interface. The item provides for the transfer to the node *S*1 (the user agrees), to the node *S*2 (the user disagrees), to the node *S*3 (the condition is not fulfilled).

Testing the item "Request a list"

This item provides a list of several records, each of which contains a number of fields (the number of records and fields is determined by the logic of the problem being solved). The output of the list may depend on some condition (Table 6).

The item provides a jump to the node *S*1 (the condition is met) or to the node *S*2 (the condition is not met).

Table 3

Template for testing the step "Create"

<table>
<tr><td rowspan="2">Output data</td><td colspan="2" rowspan="2">Input data</td><td colspan="4">Data in the system</td></tr>
<tr><td colspan="2">Entered</td><td colspan="2">There are</td></tr>
<tr><td>System message</td><td colspan="2">Object name</td><td colspan="2">Object name</td><td colspan="2">Object name</td></tr>
<tr><td rowspan="8">…..</td><td colspan="2">…..</td><td colspan="2">…..</td><td colspan="2">…..</td></tr>
<tr><td colspan="2">Parameters</td><td colspan="2">Parameters</td><td colspan="2">Conditions for receiving</td></tr>
<tr><td>Variable</td><td>Value</td><td>Variable</td><td>Value</td><td>Parameter name</td><td>Condition</td></tr>
<tr><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>……</td></tr>
<tr><td>…..</td><td>…..</td><td rowspan="4">.....</td><td rowspan="4"></td><td>…...</td><td>……</td></tr>
<tr><td colspan="2" rowspan="2">Characteristic</td><td colspan="2">Object name</td></tr>
<tr><td colspan="2">…..</td></tr>
<tr><td colspan="2">…..</td><td colspan="2">Characteristic</td></tr>
<tr><td colspan="2"></td><td colspan="2">…..</td></tr>
<tr><td colspan="2"></td><td colspan="2">…..</td></tr>
</table>

Table 4

Test templete for the step "Enter dat"

<table>
<tr><td>Output data</td><td colspan="2" rowspan="2">Input data</td><td colspan="4">Data in the system</td></tr>
<tr><td>System message</td><td colspan="3">Entered</td><td>There are</td></tr>
<tr><td rowspan="4">…..</td><td>Variable</td><td>Value</td><td>Object storing</td><td>Variable</td><td>Value</td><td>Receiving conditions</td></tr>
<tr><td>…..</td><td>…...</td><td>…..</td><td>…..</td><td>…..</td><td>…..</td></tr>
<tr><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>true</td></tr>
<tr><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>…..</td><td>…..</td></tr>
</table>

Table 5

Test template for the item "Request value"

<table>
<tr><td colspan="2">Output data</td><td>Input data</td><td colspan="2">Data in the system</td></tr>
<tr><td>Variable</td><td>Value</td><td>Variable</td><td colspan="2">Object name</td></tr>
<tr><td rowspan="4">…..</td><td rowspan="4">…..</td><td rowspan="4">…..</td><td colspan="2">…..</td></tr>
<tr><td colspan="2">Data</td></tr>
<tr><td>Variable</td><td>Value</td></tr>
<tr><td>…..</td><td>…..</td></tr>
<tr><td colspan="2">System message</td><td>Condition</td><td colspan="2">Object name</td></tr>
<tr><td colspan="2" rowspan="4">…..</td><td rowspan="4">…..</td><td colspan="2">…..</td></tr>
<tr><td colspan="2">Condition</td></tr>
<tr><td colspan="2">…..</td></tr>
<tr><td colspan="2">……</td></tr>
</table>

Table 6

Test pattern for the item "Request value"

<table>
<tr><td>Output data</td><td>Input data</td><td>Data in the system</td></tr>
<tr><td>List name</td><td>List name</td><td>Object name</td></tr>
<tr><td rowspan="2">…..</td><td rowspan="4">…..</td><td>…..</td></tr>
<tr><td>List name</td></tr>
<tr><td rowspan="2">List records</td><td>…..</td></tr>
<tr><td>List records</td></tr>
<tr><td>…..</td><td></td><td>…..</td></tr>
<tr><td>…..</td><td>Condition</td><td>Object name</td></tr>
<tr><td rowspan="2">System message</td><td rowspan="4">…..</td><td>…..</td></tr>
<tr><td>Condition</td></tr>
<tr><td rowspan="2">…..</td><td>…..</td></tr>
<tr><td>……</td></tr>
</table>

Testing the item "Choose from the list"

This item allows the user to select a record from the list (Table 7). There are two types of records: data and services (documents). Selecting a record of the first type involves receiving data by the system. Selecting a record of the second type provides a transition to the corresponding step in the scenario. If the transition is not provided, then a constant is written in the "Transition" field. User has the right to refuse to choose from the list if there is no necessary service (document) or suitable data.

The item provides for the transition to a node *S*1 or nodes *S*2, *S*3, ..., *Sn*.

ISSN 2076-2429 (print)
ISSN 2223-3814 (online)

Table 7

Test template for the item "Choose from the list"

<table>
<tr><td>Output data</td><td>Input data</td><td colspan="3">Data in the system</td></tr>
<tr><td>System message</td><td>List name</td><td>Receiving</td><td colspan="2">There are</td></tr>
<tr><td rowspan="8">…..</td><td rowspan="4">…..</td><td>Object name</td><td colspan="2">Object name</td></tr>
<tr><td>…..</td><td colspan="2">…..</td></tr>
<tr><td>Record</td><td colspan="2">List name</td></tr>
<tr><td rowspan="5">…..</td><td colspan="2">…..</td></tr>
<tr><td colspan="2">List name</td></tr>
<tr><td>Record</td><td>Record</td><td>Transition</td></tr>
<tr><td rowspan="2">…..</td><td>…..</td><td>…..</td></tr>
<tr><td>…..</td><td>false</td></tr>
</table>

Testing the item "Enter service (document)"

This item provides for the user to enter some service (document). The system compares the ordered service with the services provided from the existing list. A variant of the breakdown of the entered service into elementary services and comparison with services from the list is possible (Table 8).

Table 8

Test template for the item "Enter a service (document)"

<table>
<tr><td>Output data</td><td>Input data</td><td colspan="3">Data in the system</td></tr>
<tr><td>Service</td><td>Service (document)</td><td>Receiving</td><td colspan="2">There are</td></tr>
<tr><td>…..</td><td rowspan="7">…..</td><td>Object name</td><td colspan="2">Object name</td></tr>
<tr><td>System message</td><td>…..</td><td colspan="2">…..</td></tr>
<tr><td rowspan="6">…..</td><td>Service</td><td colspan="2">Service list</td></tr>
<tr><td rowspan="5">…..</td><td colspan="2">…..</td></tr>
<tr><td>Service</td><td>Transition</td></tr>
<tr><td>…..</td><td>……</td></tr>
<tr><td>…..</td><td>…..</td></tr>
</table>

Testing the item "Repeating actions"

The execution of the "Repeating actions" item is implemented at the system interface level. From the point of view of preparing data for testing in repeated passes, it may be necessary to increase the amount of data of some types in the system.

Software implementation and testing

In the scope of work [3], master tools were created for the automated compilation of script items using a pre-compiled vocabulary of the subject area [11, 12]. Also, there were proposed the structures of objects that can support the execution of script points. Within context of this work, software has been created for the automated compilation of adjacent lists, determination of independent paths, and providing test templates for each type of scenario item to the system analyst. The conducted experiments showed that the formation of test variants simultaneously with the setting up of the use case increases the total operating time by less than 20 %, while the compilation of tests in the offline mode is commensurate with the time of writing the use case.

Conclusions

It is proposed to combine the processes of describing the use cases and compiling case test sets. For this, a mathematical model of the use case was developed, which allows determine the sequence and completeness of testing. Based on the previously proposed classification, case-test templates for each type of item from the use case scenarios have been developed. Software tools have been developed that support the technology of developing case-test sets. The experiments carried out by the technology showed a significant reduction in time compared to existing solutions that provide separate processes for describing and compiling of test case sets.

Литература

1. Д. Леффингуэлл, Д. Уидриг. Принципы работы с требованиями. Унифицированный подход. Москва: Издательский дом Вильямс, 2002. 450с.
2. Алистер Коберн. Современные методы описания функциональных требований к системам. Москва: Лори, 2002. 266 с.
3. Возовиков Ю.Н., Кунгурцев А.Б., Новикова Н.А. Информационная технология автоматизированного составления вариантов использования. *Наукові праці Донецького національного технічного університету*. Покровськ, 2017. №1(30). С. 46–59.
4. Создание проекта. Анализ прецедентов. Реализация прецедентов. Уточненное описание прецедента. URL: http://vunivere.ru/work72704 (дата звернення 27.06.2017)
5. Куликов, С.С. Тестирование программного обеспечения. Базовый курс. Минск: Четыре четверти, 2017. 312 с.
6. Кент Бек. Экстремальное программирование: разработка через тестирование = Test–driven Development. Питер, 2003. 224 с.
7. Криспин, Лайза, Джанет Грегори. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд = Agile Testing: A Practical Guide for Testers and Agile Teams. Москва: «Вильямс», 2010. 464 с.
8. Александров А.. Тест-дизайн: проще читать или проще писать. Доклад на 15-ой SQA Days в Москве. URL: https://habrahabr.ru/company/sqalab/blog/242385/ (дата звернення 11.07.2017)
9. Каша, Андрей. Модульное тестирование: 2+2 = 4? Дата исправления: 10.12.2016 URL: ttp://rsdn.org/article/testing/UnitTesting.xml (дата звернення 17.07.2017)
10. Орлов С. Технология разработки программного обеспечения. СПб.: Питер, 2002. 464 с.
11. Кунгурцев А. Б., Поточняк Я.В., Силяев Д.Ф. Метод автоматизированного построения толкового словаря предметной области. *Технологический аудит и резервы производства.* № 2/2(22), 2015. С 58–63.
12. Кунгурцев О., Ковальчук С., Поточняк Я., Широкоступ М. Побудова словника предметної області на основі автоматизованого аналізу текстів українською мовою. *Технічні науки та технології.* 2016. № 3 (5). С. 164–174.

Referenses

1. Leffinguell, D., & Uidrig, D. (2002). *Principles of work with requirements. An unified approach*. Moscow: Publishing house Williams.
2. Alister, Kobern. (2002). Modern methods of describing of functional requirements for systems. Moscow: Lori.
3. Vozovikov, Yu.N., Kungurtsev, A.B., & Novikov, N.A. (2017). Information technology for automated generation of use cases. *Scientific papers of the Donetsk National Technical University*. 1 (30), 46–59.
4. Project creation. Analysis of precedents. Implementation of precedents. Refined description of the precedent. *vunivere.ru*. Retrieved from: http://vunivere.ru/work72704
5. Kulikov, S.S. (2017). *Software testing. Basic course.* Minsk: Four quarters.
6. Kent, Bek. (2003). *Extreme programming: development through testing = Test-driven Development.* Peter.
7. Krispin, Layza, & Janet, Gregory. (2010). *Flexible testing: a practical guide for software testers and flexible teams.* Moscow: Williams.
8. Aleksandrov, A. (2014). Test-design: easier to read or easier to write. *Report on the 15th SQA Days in Moscow*. Retrieved from: https://habrahabr.ru/company/sqalab/blog/242385/
9. Kasha, Andrew. (2016). Unit testing: 2 + 2 = 4? rsdn.org. Date of correction: 10/12/2016. Retrieved from: http: //rsdn.org/article/testing/UnitTesting.xml
10. Orlov, S. (2002). Technology of software development. St. Petersburg: Peter.
11. Kungurtsev, A. B., Potocnjak ,Ya.V., & Silyaev, D.F. (2015). The method of automated construction of an explanatory dictionary of a subject domain. *Technological audit and production reserves.* 2/2 (22), 58–63.
12. Kungurtsev, O., Kovalchuk, S., Potokonyak, Ya., & Shirokokest, M. (2016). Construction of a domain-specific dictionary on the basis of automated analysis of texts in Ukrainian. *Technical sciences and technologies.* 3 (5), 164–174.

Received September 12, 2017

Accepted October 02, 2017