BEST: International Journal of Management, Information Technology and Engineering (BEST: IJMITE)
ISSN 2348-0513
Vol. 3, Issue 2, Feb 2015, 15-26
© BEST Journals

ARCHITECTURAL REFACTORING OF MULTILAYERED PROGRAM SYSTEMS

NAZAROV STANISLAV VIKTOROVICH

Doctor of Engineering, Higher School of Economics National Research University, Moscow

ABSTRACT

An approach to the representation of structures in multilayered program systems is offered. The problem of architectural refactoring of multilayered program system for the purpose of increasing the productivity of the system is considered, while the mathematical problem definition and its decision are given.

KEYWORDS: Refactoring, Architecture, Multilayered Program System, Productivity

INTRODUCTION

The "refactoring" concept initially appeared in groups associated with Smalltalk, but soon it became popular among users of other programming languages. M. Fowler described refactoring as a small change in the source code that supports the improvement of a code project without change of its semantics.[1] He also expressed an idea in reference to "database refactoring," but architectural refactoring wasn't the true intention. It should be noted that today too few publications review problems in architectural refactoring. M. Ksenzov's publications [2,3] should be noted as the main examples. At the same time, the evolution of complex program systems requires increased attention to the selection of the architecture from a developer. Actually, new requirements are always made by a customer during development, and the initial architecture should be revised. The following stages of architectural refactoring are distinguished: 1) the "mining" of the architecture; 2) transformation of the architecture; 3) a semantic analysis of subsystems; and 4) the projection of changes on a program code. Issues on the refactoring of multilayered program systems (PS) for the purpose of improving system performance are reviewed and solved in this publication.

1. INITIAL STAGE OF PROGRAM SYSTEM DEVELOPMENT

A significant part of program systems (PS) is, as a rule, developed on an urgent basis. The integrity of interaction among business processes should be computerized (i.e, supporting PS should be developed). Frequently, the terms of reference (albeit hurriedly developed) are transferred to a chosen computer firm (possibly without a preliminary analysis or on a tender basis) that promises to perform work for the required (as a rule, the minimum) terms and for acceptable costs.

Usually, such firms apply flexible methods for the development of program systems based on the iteration and incremental approach of the PS development. This can be a SCRAM or Agile methodology with elements of extreme programming. In this case, large-scale design can be prevented so that less effort is necessary for advance design. Such an approach to PS development permits the development of integrity in program modules, thus bringing sufficient speed to computerizing the given repertoire of business processes *B*. Often, however, these modules are developed independently of each other and, in this case, the interception of functions performed by modules can occur. Situations in which one module can apply to another for the performance of some functions it would otherwise have to handle are possible (and increasing-ly frequent).

In this case it should be noted that a module means a sufficiently arbitrary structural element of the PS (e. g, a subsystem, component, separate program module, group of classes or separate class) that can be distinguished by specifying the interface of interaction between these modules and their surroundings.

Situations frequently occur in which a PS to be developed is poorly documented and developers don't specially think about the architecture and suitability of a system to be developed. Nevertheless, the architecture of the system to be developed exists, and it must be properly developed by its developers regardless of their wish. As an initial approximation, the PS architecture can be represented by a set of program modules:

$$M = \{m_{ij} \mid i = 1,2,\dots,N,\ j = 1,2,\dots,n_i\},$$

$$M = \bigcup_{i}^{N} M_i\ ,\quad |M| = K,$$

where N - quantity of business processes;

K - quantity of modules in a program system;

i - number of business processes;

j - number of a module performing the j-function of i business process;

n_i - quantity of functions performed by i business process;

M_i - subset of modules computerizing i business process $b_i \in B$.

Generally the following correlation is true:

$$\bigcap_{i}^{N} M_i \neq \emptyset.$$

Each m_{ij} module can be represented by the following parameters of specification:

$$P_{ij} = \{Name, I_{ij}, O_{ij}, A_{ij}\},$$

where *Name* - name of m_{ij} module;

I_{ij} - parameters of the input interface of m_{ij} module;

O_{ij} - parameters of the output interface of m_{ij} module;

A_{ij} - abstraction of an algorithm performed by m_{ij} module.

It should be noted that abstraction through specification allows separation from an algorithm described in a module to the level of knowledge but only in regard to the fact that it should ultimately be performed by this module. There is a representation of the $O: B \rightarrow M$ type that determines subsets of modules computerizing the functions of specific business processes. Thus there are representations $O_i\colon\ b_i \rightarrow M_i \subset M, i = 1,2,\dots,N.$ In this case, nonempty intersections are possible.

$$M_i \cap M_j \neq \emptyset, i, j = 1,2,\dots,N.$$

This shows the possible duplication of some functions of business processes in the respective PS modules to be computerized. However, it is possible to have a situation in which there is $O_i \mid M_i \mid < \mid b_i \mid$ for some representation. This means that PS performs not all functions of b_i business processes. In any event, there is no claim as to a developed program system, as was noted above. Frequently, the PS architecture is neither preliminarily developed nor sufficiently documented (perhaps not documented at all). Thus a problem of "architecture mining," as it is called in the literature,[2,3] occurs in the case of further development of a system or its maintenance.

2. REPRESENTATION OF THE DEVELOPED PS ARCHITECTURE (ARCHITECTURE MINING)

The use of graphs is a convenient visual method for the representation of the architecture of program systems. A PS model is developed on the basis of the source code, when information on units comprising the system is absent from publication.[4] In our case, an example of the PS development is the basis of a flexible method, when new modules are sequentially added in the review of a system to be developed. In this case each module m_{ij} in a system can be represented by *Name* name and a portion of the parameters O_{ij} from the specification of a module - by names of modules that can be applied from module m_{ij}. Each module will be represented as follows for the convenience and simplicity of further plotting:

$m_{ij} \rightarrow < number1, number2, number3, \dots >,$

where $number1$ - a number of module m_{ij} identified with its name *Name;*

$number2$ - number of the first module, to which a module can apply m_{ij};

$number3$ - number of the second module, to which module m_{ij} can apply, etc.

Thus, a list of all modules and their interconnections can generally be represented by the following list:

S= S_1→ S_2→⋯ →S_(K,)

where $S_i, i = 1,2, \dots, K$ - elements of a list of the following structure:

$S_1 = < 1, s, k, m, \dots >,$

$S_2 = < 2, l, t, f, \dots >,$

$S_K < K, g, z, u, \dots >,$

where $s, k, m, l, t, f, \dots, g, z, u, \dots$ - numbers of modules.

G graph representing the PS structure can be plotted on the basis of *S* list. However, with this representation it is difficult to make a conclusion regarding a type of program system architecture and its quality. It is known that significant portions of up-to-date program systems have multilayered architecture. Modules of lower layers don't apply to other layers in order to perform functions in such architectures, and consequently the arrangement of upper layers can be different. That is why the first problem is the separation of module layers during the analysis of obtained PS architecture. The multilayered architecture provides for the grouping of bound functionality of an application in different layers vertically built above each other. The functionality of each layer is joined by the common role or responsibility: Layers are loosely linked, and clear data exchange is performed between them. The correct division of an application into layers supports the strict allocation of functionality, thus providing flexibility as well as the convenience and simplicity of development.

Application layers can be physically located in one computer (at one level) or can be distributed between different computers (n-levels). The interlinking of components of different levels is performed by means of strictly specified interfaces. Accordingly, we will review the PS of one specific language level with properly determined syntactic units according to definite syntactic rules and the semantics of elementary operators and syntactic constructions. All issues with respect to other language levels, such as the interpretation of elementary operators in terms of more primitive components, are omitted from this review.

Elementary operators of this language level will be reviewed as modules of the basic level comprising the basic layer. It is possible to perform this, as all elementary operators of the PS are accessible everywhere. It should be noted that privileged commands of the machine language for use in application program systems aren't taken into account in this case. Modules constructed from basic-level modules can be reviewed only as zero-level modules. In this case, equal accessibility of all zero level modules isn't essential. For example, some zero-level modules can be allocated in a block, whereupon they are accessible only within that block and its sub-blocks in programs written in a language allowing the block structure. At the same time, modules of different levels and even of the same level, as for a module to be constructed (recurring, co-routines), can be used in some languages during the construction of higher-level modules.

A matrix R of $K \times K$size should be represented in this review. Each element of this matrix is formed according to the following rule:

$$r_{ij} = \begin{cases} 1, & \textit{если } j \in S_i, \\ & 0 - if\ not. \end{cases}$$

Thus it is possible to follow an algorithm as shown below:

- Beginning, I = 0
- Number of strings with all elements equal to zero should be found in a matrix.
- Peaks with these number forming I-layer should be fixed.
- I = I + 1
- If columns with non-zero elements remain, columns with numbers of found peaks should be zeroed out. Go to point 1.
- If all columns contain only zero element, this is the end.

It should be noted that this algorithm allows the construction of the layer-by-layer architecture of the PS that meets one of the variants reviewed in reference 5. However, if there are horizontal links or closely coupled modules in PS layers it is impossible to accurately determine the PS structure without additional analysis.

3. ANALYSIS OF COMPLIANCE WITH THE LAYER-BY-LAYER ARCHITECTURE (SEPARATION OF LAYERS)

Let us review a procedure of analysis performance with a specific example. Some integrity of the PS modules (nine modules) represented by the following R matrix is set by S list.

$$R = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Null strings are found according to the above note algorithm, and zero-level modules are determined in such way: $L_0 = \{8, 9\}$. The columns 8 and 9 are eliminated. Zero string with number 7 determining layer $L_1 = \{7\}$ should be found. The seventh column should be eliminated, and layer $L_2 = \{4, 5\}$ should be determined. Columns 4 and 5 should be eliminated. Modules of the third layer $L_3 = \{2, 3, 6\}$ should be determined according to the remaining zero strings. The strings 2, 3 and 6 should be eliminated. Columns with these numbers should be eliminated, and modules of the fourth layer $L_4 = \{1\}$. should be determined. After the distribution of modules in PS layers, a *G* graph of the program system can be plotted (Figure 1).

It should be noted that, while analyzing a graph thus obtained, it doesn't satisfy the classical rules of a multi-layered structure. Namely, module 6 doesn't meet the requirements. It is known that the separation of layers is a good basis for system improvement. It is somewhat difficult to find layers in an arbitrary program system, because they can include horizontal links and closely coupled components, as has previously been noted. That is why it's reasonable to expand a "layer" notion to allow the inclusion of closely coupled components [2,3] in arbitrary layers.

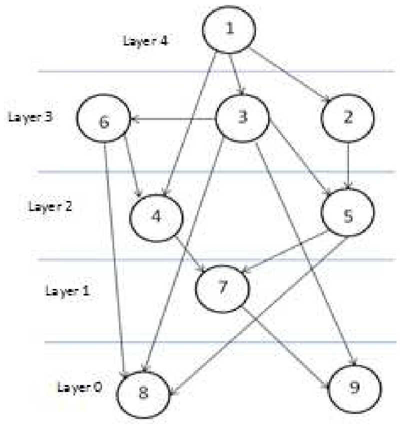

Figure 1: PS Graph

These components can be reviewed as atomic modules through means of this approach. It should be noted that closely coupled components not always show poor architecture of a system. A possible defect of the architecture with absorption layers can be the "lost layer" effect, in which a defective link causes occurrence of modules that should be allocated in different layers as to their meanings.

4. CORRECTION (TRANSFORMATION) OF THE ARCHITECTURE FOR ITS REFACTORING

Refactoring is based on the sequence of small, equal (in other words, the remaining characters) transformations that keep the functional semantics of the source code. Frequently, a problem involving the discovery of a meaning of modules occurs during transformations. A task for an expert involved in the architectural refactoring process is to make the semantic analysis volume as low as possible (e.g, by deleting the auxiliary units) and to make it consistent and directional. Such a code change that has no impact on the structure of modules, i.e, a program code within classes that is subjected to refactoring, can be considered as the first level of refactoring. The second level of refactoring belongs to the change of the structure of the modules or classes of a program system, the addition of new classes, the separation and division of large classes, the transmittal or addition of new methods, the separation of interfaces, etc.

The next, third level of refactoring is referred to by M. Fowler as large refactoring. The monograph [1] reviews four cases of refactoring at the third level: division of the extension, the transformation of a procedural design into objects, the separation of the object domain from a conception, and the determination of the hierarchy. The architectural refactoring of program systems is the fourth level of refactoring. The need for architectural refactoring is attributed to the following reasons:

- A poorly structured code due to frequent changes made by developers that do not fully understand PS architecture.
- Improvement of the PS performance: The refactoring of the first and second levels undoubtedly forces a program to operate more slowly, but in this case it makes the program clearer and more flexible for performance setting.
- Need for functional changes of the PS: A change in the existent architecture can be a good step toward the implementation of new functionality that facilitates the further evolution of a system.
- Change of a PS platform: It is preferable this be limited to changes only in a narrow, platform-dependent interlayer of a system. The separation of such an interlayer is always associated with the need to change the architecture.
- The upgrade of a method for the development of a software product associated, for example, with transferring to a more improved programming method through the implementation of a complex environment of collective development, etc.
- Transformations associated with the reorganization of a company performing the development. For example, outsourcing performance. The change of the PS architecture can simplify the decision in this task.

Prior to the review and correction of the architecture, it is necessary to determine a means to evaluate the quality of the PS structure. It is known from engineering experience that the best decision is provided by the hierarchy structure of a tree type. The degree of difference of the real design structure from a tree is specified by the disparity structure. It is known that the full graph with n peaks has a quantity of edges equal to $e_c = n \times (n-1)/2$, while a tree with the same quantity of peaks has a significantly smaller quantity of edges $e_t = n - 1$. A disparity formula can be developed through comparison of the quantity of edges in the full graph, real graph and tree. Disparity for the design structure with n peaks and e edges should be determined according to the following formula:

$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}$$

The disparity value is included in the range from 0 to 1. If *Nev* = 0, the design structure is a tree, if *Nev* = 1, the design structure is the full graph. Clearly, the disparity gives the rough estimate of a structure. In order to improve the accuracy of the estimate, the characteristics of connection and bonding should be applied.[4]

Let us return to the structure given in Figure 1. It is clear that module 6 cannot be located in the same layer as module 3. Given the fact that module 3 applies to module 6 in order to perform its function, the latter should be transferred to a lower layer. A possible variant for a new structure is shown in Figure 2. It should be noted that module 2 would, in this case, be transferred to the lower layer 3. Attention should be paid to the increased number of the PS layers after correction is performed. This important factor can cause increasing of operation time of the PS. Additionally, it should be noted that PS complexity determined according to *Nev* value isn't changed after such correction of a system, because the quantity of peaks and edges is the same.

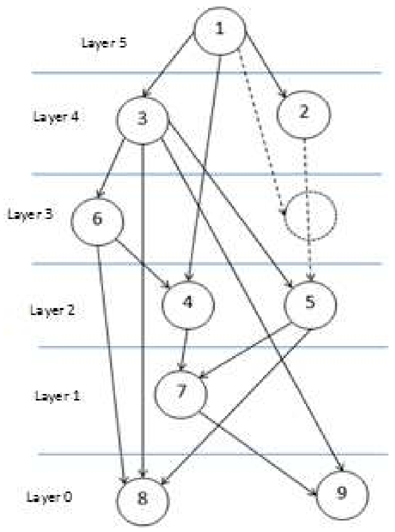

Figure 2: First Variant of Correction of the PS Architecture

Another variant of architecture correction is possible. It is associated with the integration of modules 3 and 6 (modules 3 through 6 in Figure 3). In this case the quantity of the PS layers isn’t changed, but the number of peaks and edges (8 peaks and 12 edges) is changed. This slightly reduces the complexity of the PS as to the value *Nev*. However, an integrated module increases as to volume and its programming is complicated.

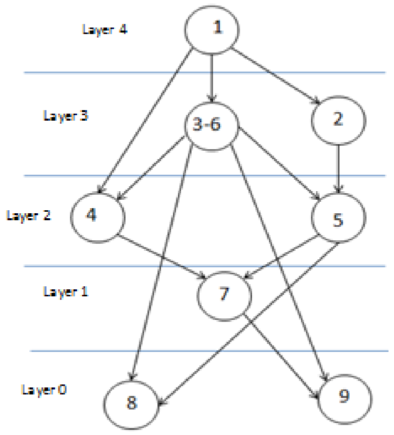

Figure 3: Second Variant of Correction in the PS Architecture

It is very difficult to formalize a process of correction of the PS architecture or to develop an algorithm of correction moreover. However, in a number of cases, by distinguishing the separate fragments of the PS structure they can be transformed in order to obtain the best structure, such as a tree. Most often it can be done through the integration (absorption) of modules. Examples of such correction are given below:

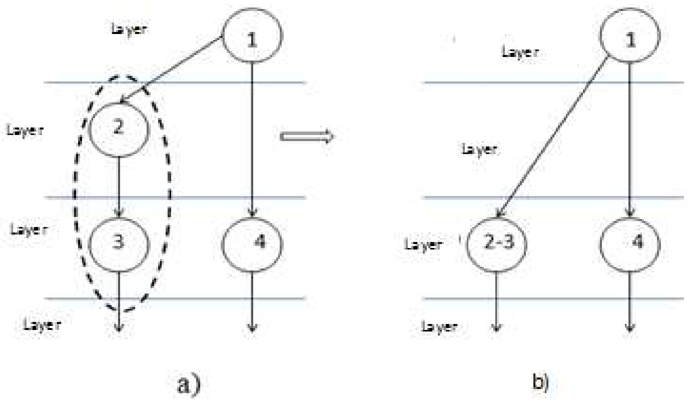

Figure 4: Correction through Absorption by a Lower Level

A serial chain of modules 2 and 3 used by module 1 only is shown in Figure 4a. It might occur due to the attempt to parallelize the operation for the programming of these modules. A possible variant for improvement of the PS structure by the integration of modules 2 and 3 is shown in Figure 4b. It should be noted that if, in this case, module 1 applies to modules 2, 3 and 4 only, the quantity of the PS layers will be reduced.

A case in which the results of operation of modules 1 and 2 are used by module 3 only is shown in Figure 5a. The PS structure can be improved with the integration of modules 1 and 2 as shown in Figure 5b.

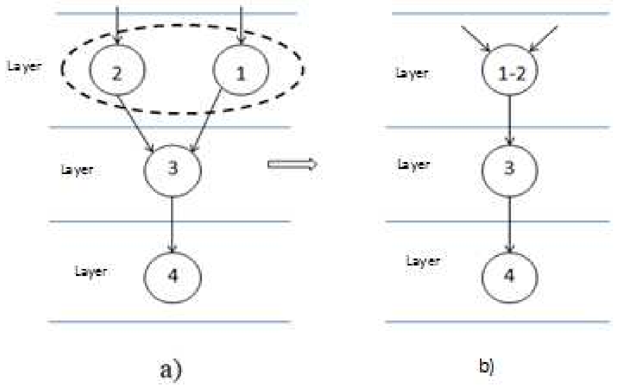

Figure 5: Correction by Integration (Variant 1)

Another case of integration of modules 2 and 3 is given in Figure 6a. It is possible, when module 1 applies to modules 2 and 3, and in turn modules 2 and 3 apply to module 4. A PS structure is simplified due to the integration of modules 2 and 3, as shown in Figure 6b.

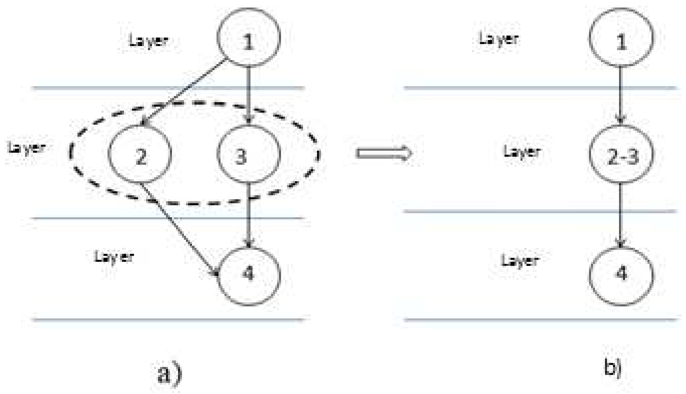

Figure 6: Correction by Integration (Variant 2)

Situations of division of modules, as shown in Figure 7, are possible in the correction of the PS architecture in addition to the integration of modules and transference by layers. However, a decision on this or that correction of the PS structure should be taken after its detailed analysis and expediency consideration in each particular case.

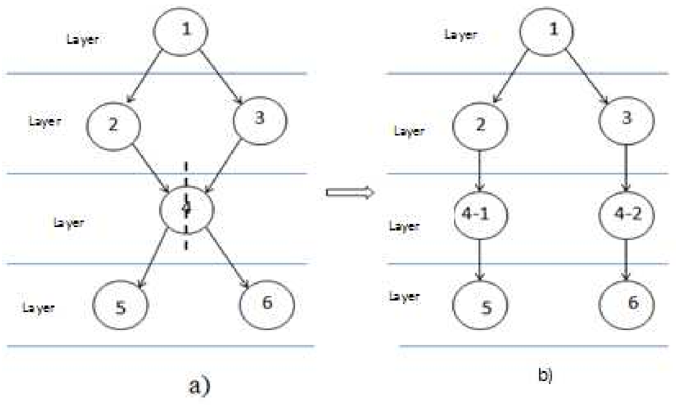

Figure 7: Correction by Division

The result of architecture correction should be projected to the real program code of a system. A set of strings and files that meet a remote unit in a program code should be determined for the projected removal of modules from a model. Subsequently, the strings and files thus found should be deleted from the program design. The corresponding strings and files in the source code of the program system are transferred and projected to a code of module transference in a model, etc. Transformations performed in such a way can be reviewed as the architecture - driven refactoring of a program code.

5. ARCHITECTURAL REFACTORING IN ORDER TO INCREASE PS PERFORMANCE

5.1. Variants of Multilayered Structures

Let us review the variants of development in a preliminary, multilayered PS. Two possible general structures for the arrangement of program layers are shown in the figures 8 and 9. The notion of the PS layers is associated with the concept of multilevel virtual machines. Figure 8 shows an approach used when a task of program architecture development is reviewed as the development of the "user's machine" or a virtual machine (n), starting from the lowest level (0) of hardware (or possibly the operating system). The sequence of levels referred to as abstract machines is specified in such a way that each successive machine is developed on the basis of the previous machine, thereby extending their capabilities. Each level can refer to one level different from its own only, namely to the level that immediately preceded it.

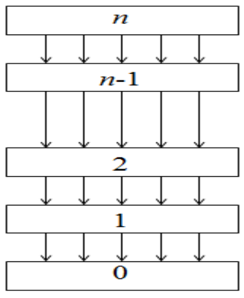

Figure 8: A Variant of the Classic Architecture

Levels aren't full abstracts of lower level, but each of them can refer to all preceding levels in the structure shown in Figure 9. Moreover, a third variant is possible as an intermediate between the first two. In this case the use of only some commands provided by layers (1), (2)..., (i-1) is allowed for layer (i). Each variant has own advantages and disadvantages.

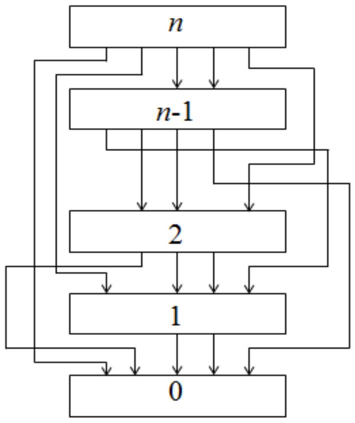

Figure 9: A Variant of the Structure with References to All Previous Layers

Let us review the features of the main variants in multilayered structures. If each layer has access to commands of just one layer in the variant of Figure 8, the developer should take into account only the previous layer. However, while this variant seems to be attractive from the perspective of engineering, it can be very ineffective. For example, if some tool represented by layer (2) will be required by layer (i), each of the layers (3), (4)..., (i-1) should provide for this tool. It means that the query of this tool by layer (1) should "sink" down through layer (i-1) till the achievement of layer (2), which is able to execute the query. Such an approach is associated with additional time consumption for query transmittal. These difficulties associated with the problem of effectiveness can force one to apply the structure in Figure 2, where each layer (i), wherein 2< i < n, can directly apply to layer (2).

Accordingly, the task of determining the optimal structure of a multilayered PS becomes very important from the perspective of performance.

5.2. Statement of Work

Let us imagine the generalized structure of a multilayered program system represented in Figure 10. In this case each layer is shown as a separate module with the ability to arrange links with any arbitrary layer in a system. Such a generalized graph permits the review of any structure of an *n*-layer program system within the range of structures represented in the figures 8 and 9. An arbitrary structure is described by some set of Boolean variables

$$X = \{x_{ij} \in \{0,1\} | \ i = n, n-1, \dots, 2; j = n-1, n-2, \dots, 1; i > j\}. \quad (1)$$

where $x_{ij} = 1$, if there is a link between the layers i and j, and $x_{ij} = 0$, if there is no such link. As a link always exists between adjacent layers, thus

$$(\forall i \ | i = j+1) \ (x_{ij} = 1), i = n, n-1, \dots, 2. \quad (2)$$

If only variables described by statement (2) take the unit value in the multilayered structure of a program represented by expression (1), this program has a structure corresponding to the variant in Figure 8. If the following statement is true

$$(\exists \ i \ | (i = n, n-1, \dots, 2)) \& (\exists \ j | (i - j \geq 2)) (x_{ij} = 1), \quad (3)$$

A program has a structure corresponding to an intermediate variant between the variants of the structures shown in the figures 8 and 9.

If the following statement is true

$$(\forall\, i \mid i = n, n-1, \ldots, 2)\ \big(\exists\, j \mid (i - j \geq 2)\big)\big(x_{ij} = 1\big), \tag{4}$$

A program has a structure corresponding to a variant shown in Figure 8.

It is convenient to perform the further setting of a problem according to a specific example. Given the fact that quantity of layers doesn't exceed three to five for most of the existing programs, let's review a five-layer program having the structure represented by Figure 10. In this case the structure of a developed PS has a variant of the classic architecture, i.e, it meets statement (2). Possible additional links between PS layers are shown by dash lines in Figure 10. A variable, which unit value means availability of interlayer ling and zero - absence of this link, is set for each line in compliance.

It is considered that PS has passed the full testing stage and temporary characteristics of modules are specified in the debugging process. Additionally, the frequency of application of modules from an arbitrary layer to modules from lower layers is specified. Let us consider that transmittal (translation) of query through i layer downloads this layer for some time interval t_i additionally (in addition to the performance of proper functions). If $x_5 = 0$, (i.e, there is no link between the layers 5 and 3), m_4 module is additionally active for time period t_5. If $x_5 = 1$ (i.e, a link between the layers 5 and 3 is set), the m_4 module doesn't require an additional time period. However, in this case it is necessary to add an intermodule interface for the interaction of the m_5 and m_3 modules. Let us consider that it enlarges a program in some value e_5. Analogous arguments are also true for other variables, as are shown in Figure 10.

Accordingly, additional links between program layers reduce the time needed for the performance of its functions by increasing the size of the program. Additionally, it is necessary to consider the fact that additional links to be placed between layers can operate with different loads.

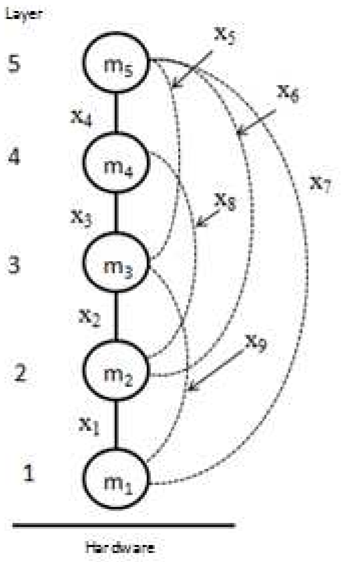

Figure 10: Structure of a Five-Layer Program

Thus, for example, if a link indicated by x_5 variable is to be developed, the m_4 module becomes free only from the transmittal of such queries that are addressed by the m_5 module to the m_3 module. That is why it's reasonable to put, for each variable x_i , a specific intensity of interaction of some pair of modules $\lambda_i$in compliance. Therefore, the architectural refactoring of the PS is reduced to the determination of such structure of a multilayered program system that provides for the best performance of a program at set limits of size for additional intermodule interfaces.

5.3. Mathematical Setting of the Problem and its Solution

In our case the structure of a multilayered program can be represented by the vector $X = \{x_i | \ i = 5, 6, \dots, 9\}$ (it should be noted that $x_1 = x_2 = x_3 = x_4 = 1$ always, as these variables determine the links between adjacent layers). That is why it's necessary to find such value X_{opt} that provides for the maximum benefit in time of the PS operation

$$Max\ T = \sum_{i=5}^{i=9} \lambda_i \times t_i \times x_i, \tag{5}$$

if the limit for permissible enlargement of E program, since the additional intermodule interfaces is observed

$$\sum_{i=5}^{i=9} e_i \times x_i \ \leq E. \tag{6}$$

Taking into account the binary character of variables, the following limit should be added:

$$\forall\ i\ (x_i\ \in \{0,\ 1\}). \tag{7}$$

A set problem belongs to the class of linear programming problems with Boolean variables (i.e, the knapsack problem). The small size of a reviewed problem allows its easy solution with the full search of variable sets representing the permissible solutions under accepted limits.

CONCLUSIONS

It is reasonable to apply the proposed approach to the architectural refactoring of the PS in the final system debugging stage, when quantitative values of time and frequency parameters of operation of modules are obtained. The size of a problem can increase significantly in real program systems containing several modules in each layer, and it will require the application of more complex algorithms of problem solution. However, due to the probabilistic character of basic data in a reviewed task, there is no point in applying methods intended to obtain an optimal decision, but it is possible to be limited to approximate, quick operating algorithms. The result of a decision should be projected onto the real program code of a system.

REFERENCES

1. Fowler M., Beck K, Brant D, Roberts D, Updike U. Refactoring: improving the design of existing code. - Spb: SImvol-Plus, 2009, p. 432.
2. Ksenzov M. Refactoring of software architecture: separation of layers Publications of the System Programming Institute of RAS, preprint 4, 2004, p. 211 - 227
3. Ksenzov M. V. Refactoring of software architecture. http://www.ispras.ru/ru/proceedings/docs/2004/8/1/isp_2004_8_1_211.pdf
4. Mironov V. O. Application of graphs for the analysis of complex systems in the basis of the source code of programs. http://berestneva.am.tpu.ru/Papers/KONF2009/%
5. Nazarov S. V. Architecture and engineering of program systems. M.: INFRA-M, 2013, p. 352.