УДК 004.05(075.8)

І.Є. Андрущак, Ю.Я. Матвіїв
Луцький національний технічний університет

ТЕХНОЛОГІЧНІ ПРИЙОМИ СТРУКТУРНОГО ПІДХОДУ ДО ПРОГРАМУВАННЯ І ЙОГО ОСНОВНІ КОНЦЕПЦІЇ

В роботі проведено огляд та розглянуті основні етапи розвитку технологій програмування та проблеми, що виникають при розробці складних програмних систем, аналізуються підходи та послідовність розробки програмного забезпечення. Виділено ключові площини прийому структурного підходу до програмування і його основних концепцій: низхідній розробці, структурному і модульному програмуванню, а також наскрізного структурному контролю.

***Ключові слова**: програмування, структурне і модульне програмування.*

И.Е. Андрущак, Ю.Я. Матвеев
Луцкий национальный технический университет

ТЕХНОЛОГИЧЕСКИЕ ПРИЕМЫ СТРУКТУРНОГО ПОДХОДА К ПРОГРАММИРОВАНИЮ И ЕГО ОСНОВНЫЕ КОНЦЕПЦИИ

В работе проведен обзор и рассмотрены основные этапы развития технологий программирования и проблемы, возникающие при разработке сложных программных систем, анализируются подходы и последовательность разработки программного обеспечения. Выделены ключевые плоскости приема структурного подхода к программированию и его основных концепций: нисходящей разработке, структурном и модульному программированию, а также сквозного структурном контроля.

***Ключевые слова:** программирование, структурное и модульное программирование.*

I.Ye. Andruschak, Yu.Ya. Matviiv
Lutsk National Technical University

TECHNOLOGICAL ADJUSTMENTS OF THE STRUCTURAL APPROACH TO PROGRAMMINGAND HIS MAIN CONCEPTS

The paper reviews and discusses the main stages of the development of programming technologies and problems arising in the development of complex software systems, analyzes the approaches and sequence of software development. The key areas for the adoption of a structured approach to programming and its main concepts are identified: descending development, structural and modular programming, as well as through structural control.

***Keywords:** programming, structural and modular programming.*

Formulation of the problem. Most modern software systems are objectively very complex. This complexity is caused by many reasons, the main of which is the logical complexity of the tasks they solve.

While the computing facilities were small and their capabilities were limited, computers were used in very narrow areas of science and technology, and, first of all, where the tasks to be solved were well-defined and required significant calculations. Nowadays, when powerful computer networks are created, it became possible to shift to them the solution of complex resource-intensive tasks, which nobody previously thought of computerizing. Now in the process of computerization completely new subject areas are involved, and for already mastered areas already established statements of tasks become more complicated.

Additional factors that increase the complexity of developing software systems are:

- the complexity of formal definition of requirements for software systems;
- the lack of satisfactory means of describing the behavior of discrete systems with a large number of states under a nondeterministic sequence of input actions;
- collective development;
- the need to increase the frequency of codes.

Setting up tasks. The complexity of determining the requirements for software systems is determined by two factors. First, when determining the requirements, it is necessary to take into account a large number of different factors. Secondly, the developers of software systems are not specialists in automated subject areas, and experts in the subject area, as a rule, can not formulate the problem in the proper foreshortening.

Absence of satisfactory means of formal description of the behavior of discrete systems. In the process of creating software systems, languages of relatively low level are used. This leads to early details of operations in the process of creating software and increases the volume of descriptions of the products

being developed, which, as a rule, exceeds hundreds of thousands of operators of the programming language. The means, which make it possible to describe in detail the behavior of complex discrete systems at a higher level than the universal programming language, do not exist.

Analysis of recent researches and publications. Due to the large volume of projects, software development is conducted by a team of specialists. Working in a team, individual professionals must interact with each other, ensuring the integrity of the project, which is difficult to achieve in the absence of satisfactory means of describing the behavior of complex systems mentioned above. Moreover, the larger the team of developers, the more difficult it is to organize the work process. The complexity of the software being developed is also affected by the fact that to increase productivity, companies are striving to create component libraries that could be used in further development. However, in this case, the components have to be made more universal, which ultimately increases the complexity of the development.

Taken together, these factors significantly increase the complexity of the development process. However, it is obvious that they are all directly related to the complexity of the development object - the software system.

Basic material presentation. Programming is a relatively young and rapidly developing branch of science and technology. The experience of conducting real developments and improving existing software and hardware is constantly being reinterpreted, resulting in new methods, methodologies and technologies that, in turn, serve as the basis for more modern software development tools. It is advisable to study the processes of creating new technologies and determine their main trends by comparing these technologies with the level of development of programming and the peculiarities of software and hardware available to programmers.

The technology of programming is called the set of methods and tools used in the software development process [1, 2]. Like any other technology, programming technology is a set of technological instructions, including:

- indication of the sequence of technological operations;
- enumeration of conditions under which this or that operation is performed;
- descriptions of the operations themselves, where for each operation, the initial data, results, as well as instructions, standards, standards, criteria and methods of assessment, etc. were determined.

In addition to the set of operations and their sequence, the technology also determines the way of describing the projected system, more precisely the model used at a particular stage of development.

Distinguish the technologies used at specific stages of development or for the solution of individual tasks of these stages, and technologies that cover several stages or the entire development process. At the heart of the former, as a rule, lies a limitedly applicable method that allows solving a specific problem. The second is usually based on a basic method or approach that defines the set of methods used at different stages of development, or methodology.

When designing, implementing, and testing the components of the structural hierarchy obtained during the decomposition, two approaches are used:

- Ascending;
- downward.

In the literature, there is another approach, called the "expansion of the nucleus." He assumes that first of all they design and develop some basis - the core of the software, for example, the data structures and procedures associated with them. In the future, the nucleus is increased by combining the ascending and descending methods. In practice, this approach, depending on the level of the kernel, reduces to either a descending or an ascending approach.

Ascending approach. When using the bottom-up approach, first design and implement the components of the lower level, then the previous one, etc. As the testing and debugging of the components are completed, they are assembled, and the lower-level components are often placed in component libraries with this approach.

To test and debug components, special testing programs are designed and implemented. The approach has the following disadvantages:

- increase in the probability of inconsistency of components due to incompleteness of specifications;
- availability of costs for the design and implementation of testing programs that can not be converted into components;

- the later design of the interface, and accordingly the inability to demonstrate it to the customer to clarify the specifications, etc.

Historically, the ascending approach appeared earlier, which is due to the peculiarity of the thinking of programmers, who in the process of learning get used to writing small programs first to detail the components of the lower levels (subroutines, classes). This allows them to better understand the processes of the upper levels. With the industrial manufacture of software, the bottom-up approach is currently practically not used [3].

Top-down approach. A top-down approach assumes that the design and subsequent implementation of components is performed "top-down", i.e. first project components of the upper levels of the hierarchy, then the next and so on down to the lowest levels. In the same sequence, the components are also implemented. At the same time, during the programming process, the components of the lower, not yet implemented levels are replaced with specially designed debugging modules - "stubs", which allows testing and debugging the already implemented part.

When using the top-down approach, hierarchical, operational and combined methods of determining the sequence of design and implementation of components are used.

The hierarchical method assumes that the development is strictly level-based. Exceptions are allowed if there is a data dependency, i.e. if it is found that some module uses the results of the other, then it is recommended to program it after this module. The main problem of this method is a large number of rather complicated plugs. In addition, using this method, the bulk of the modules are developed and implemented at the end of the project, which makes it difficult to allocate human resources.

The operational method associates the execution sequence when the program is started. The application of the method is complicated by the fact that the order of execution of the modules can depend on the data. In addition, the output modules of the results, despite the fact that they are called up last, should be developed one of the first, so as not to design a complex stub that provides the output of the results during testing. From the point of view of the distribution of human resources, it is difficult to begin work until all the modules that are on the so-called critical path are completed.

The combined method takes into account the following factors that affect the development sequence:

- reachability of the module - availability of all modules in the chain of calls for this module;
- Data dependency - modules that form some data must be created before processing;
- ensuring the possibility of issuing results - output modules should be created before processing;
- availability of auxiliary modules - auxiliary modules, for example, file closing modules, program termination, must be created before processing;
- availability of necessary resources.

In addition, all other things being equal, complex modules must be developed before simple ones, as they may exhibit inaccuracies in the specifications, and the sooner this happens, the better.

A top-down approach allows for a disruption of the downstream sequence of component development in specially specified cases. So, if some component of the lower level is used by many components of higher levels, then it is recommended to design and develop earlier than the components causing it. And, finally, first of all, design and implement components that ensure the processing of the correct data, leaving the processing components of incorrect data for the end.

A top-down approach is usually used in object-oriented programming. In accordance with the recommendations of the approach, first design and implement the user interface of the software, then develop classes of some basic objects of the domain, and only then, using these objects, design and implement the remaining components [4].

In some cases, the coupling of modules can be reduced by removing unnecessary links and structuring the necessary connections. An example is object-oriented programming, in which instead of a large number of parameters, the method implicitly receives the address of the region (structure) in which the fields of the object are located, and explicitly additional parameters. As a result, the modules are linked by a pattern.

Connectivity is a measure of the strength of the connection between functional and information objects within a single module. If the coupling characterizes the quality of separation of modules, the connectivity characterizes the degree of interconnection of the elements realized by one module. The placement of strongly coupled elements in one module reduces the intermodule links and, accordingly, the mutual influence of the modules. At the same time, putting strongly interconnected elements in different modules not only enhances intermodule communications, but also complicates the

understanding of their interaction. The combination of loosely coupled elements also reduces the adaptability of modules, since such elements are more difficult to mentally manipulate.

With functional connectivity, all objects of the module are designed to perform one function: operations that are combined to perform one function, or data associated with a single function. A module whose elements are functionally connected has a clearly defined goal, when it is called, a single task is executed, for example, a subroutine for finding the minimum element of the array. Such a module has the maximum connectivity, the consequence of which are its good technological qualities: ease of testing, modification and maintenance. It is with this that one of the requirements of structural decomposition is "one module - one link between modules-resource libraries". For example, if you design an editing function for a text editor, it is better to organize a library of editing functions than to place part of the functions in one module and part of it in another.

With a sequential connection of functions, the output of one function serves as the initial data for another function. Typically, such a module has one entry point, i. E. implements one subroutine that performs two functions. It is believed that the data used by the sequential functions are also linked in series. A module with a serial connection of functions can be divided into two or more modules, both with sequential and with functional connectivity. Such a module performs several functions, and consequently its processability is worse: it is more difficult to organize testing, and when performing a modification, one mentally divides the functions of the module.

Information-related functions are considered that process the same data. When using structured programming languages, separate execution of functions can be performed only if each function is implemented by its subroutine.

Despite the combination of several functions, the information-related module has good performance indicators. This is because all functions that work with some data are gathered in one place, which allows you to adjust only one module when changing the data format. Information related is also considered data that is processed by one function.

Functionally linked functions or data that are part of a single process. Normally, modules with procedural connection of functions are obtained if functions of alternative parts of the program are combined in the module. With procedural coherence, the individual elements of the module are extremely weakly connected, since the actions they perform are related only to the general process, hence the manufacturability of this type of communication is lower than the previous one.

Temporary connectivity of functions implies that these functions are performed in parallel or for a certain period of time. Temporary data connectivity means that they are used in a certain time interval. For example, the temporary connectivity has the functions performed when initializing a certain process. A distinctive feature of the temporal connection is that the actions realized by such functions can usually be performed in any order. The content of a module with a temporary connectivity of functions tends to change: it can include new actions and / or exclude old ones. The high probability of modifying the function further reduces the performance of the modules of this type in comparison with the previous one.

A logical connection is based on the union of data or functions into one logical group. An example is text processing functions or data of the same type. A module with logical connectivity of functions often implements alternative versions of one operation, for example, addition of integers and addition of real numbers. From this module, one of its parts will always be called, while the calling and called modules will be linked by management. Understand the logic of the modules containing logically connected components, as a rule, more complicated than modules that use temporary connectivity, therefore, their performance indicators are even lower [5].

In the event that the link between the elements is small or absent, they consider that they have random connectivity. The module, whose elements are randomly connected, has the lowest performance indicators, since the elements combined in it are not connected at all.

In the three penultimate cases, the connection between several subroutines in the module is due to external causes, and in the latter, there is no connection at all. This is appropriately projected onto the technological characteristics of the modules

As a rule, with well-designed decomposition, the modules of the upper levels of the hierarchy have functional or consistent connectivity of functions and data. Data service modules are characterized by the informational connectivity of functions. The data of such modules can be related in different ways. So, the modules containing the description of classes under the object-oriented approach are characterized by the informational connection of methods and the functional connection of data. Obtaining in the process of decomposition of modules with other types of connectivity, most likely, means insufficiently thought-out design. The only exceptions are resource libraries [6].

There are two types of resource libraries: subroutine libraries and class libraries.

Libraries of subprograms implement functions that are similar in purpose, for example, a library of graphical information output. The connectivity of subprograms among themselves in such a library is logical, and the connectivity of the subroutines themselves is functional, since each of them usually implements one function [7].

Class libraries implement close-by-purpose classes. The connectivity of class elements is informational, the connectivity of classes among themselves can be functional - for related or associate classes and logical - for the rest.

As a means of improving the technological characteristics of resource libraries, the division of the module body into the interface part and the implementation area is now widely used [8].

The interface part in this case contains a set of resource declarations (subroutine headers, variable names, types, classes, etc.) that this library provides to other modules. Resources that are not advertised in the interface are not accessible from the outside. The implementation area contains the subprogram bodies and, possibly, the internal resources (subprograms, variables, types) used by these subroutines. With such an organization, any changes in the implementation of the library that do not affect its interface do not require revision of the modules associated with the library, which improves the technological characteristics of the library modules. In addition, such libraries, as a rule, are well-established and thought out, since they are often used by different programs.

Conclusion. Creating a software system is a very time-consuming task, especially in our time, when the amount of software code exceeds hundreds of thousands of operators. A future software development specialist should have an understanding of the methods for analyzing, designing, implementing and testing software systems, as well as focusing on existing approaches and technologies.

References

1. Zhogolev E.A. Programming technology / E.A. Zhogolev. - M.: The scientific world, 2004. - 216 p.

2. Ivanova G.S. Programming technology: a textbook for universities / GS. Ivanova. - M.: Izd-vo MSTU them. N.E. Bauman, 2002. - 320 with.

3. Lippman S.B. Fundamentals of programming in C ++ / SB. Lippmann. - M.: Williams, 2002. - 256 p.

4. Shieldt G. C / C ++. Reference book of the programmer / G. Shildt. - M.: Williams, 2000. - 448 p.

5. Prata S. The programming language of C ++. Lectures and exercises / S. Prata. - M.: DiSoft, 2005. - 645 p.

6. Podbelsky V.V. C ++ language / V.V. Podbelsky. - M.: Finance and Statistics, 2003. - 562 p.

7. Straustrup B. The programming language of C ++. Special edition / B. Straustrup. - M.: Binom-Press, 2008. - 1104 p.

8. Davydov V.G. C ++ Programming Technologies / V.G. Davydov. - St. Petersburg. : BHV-Petersburg, 2005. - 672 p.