Scientific Research Publishing

Journal of Software Engineering and Applications, 2018, 11, 408-434
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Scientific Basis of System Programming

E. M. Lavrischeva

ISP RAS, Moscow, Russia
Email: lavryscheva@gmail.com, lavr@ispras.ru

How to cite this paper: Lavrischeva, E.M. (2018) Scientific Basis of System Programming. *Journal of Software Engineering and Applications*, **11**, 408-434.
https://doi.org/10.4236/jsea.2018.118025

Received: July 9, 2018
Accepted: August 27, 2018
Published: August 30, 2018

Abstract

Theoretical foundations of programming systems from modules, objects, components, services are given. Identified relevant theory of programming proposed by the author with the participation of students and postgraduates: graph modular programming theory with graph representation as an adjacency matrix for mathematical achievability of graph vertices; theory of generating programming and theory of software factories; theory of graph object and component modeling (OCM) by means of logic and algebra-mathematical theory of determining individual elements of complex systems; theory of system programming based on ontological and service-component models (SOA, SCA) with security and quality systems. The Internet Smart and Nanotechnology are given for perspective transition of computer technology to nanotechnology.

Keywords

Science, Concepts, Formalism of Modules, Interface, Assembly Method, Object, Component, Life Cycle, Logic-Mathematical Theory, Model OM, FM, Configuration, Verification, Testing, Reliability, Quality Products

1. Introduction

The author of the article is a mathematician by education and for more than 50 years has been engaged in the development of the idea of A. P. Ershov (1972) in the future to develop the theory of programming as a mathematical science [1]. Since the advent of different programming languages (LP) to describe different kinds of programs for solving mathematical, biological, economic and other problems with the help of electronic computers, many scientists began to develop separate areas of programming theory. The theory includes formal descriptions of programs, their translation, debugging, testing and quality assessment. In connection with the application of international LP (Algol, PL/1, Fortran,

Cobol, Prolog, Smalltalk, Module, etc.) for describing different types of modules that implement the functional tasks of the computer of type IBM-360. In 1975, the idea of an Assembly of diverse modules, like assembling a car from finished parts in the factories of Ford. Some of the theories programming are discussed below.

The author has formulated the method of Assembly of modules based on interfaces (intermodule and interlanguage) and was published in articles [2] [3] [4]. In the method of the Assembly is implemented the theory and practice of transforming non-equivalent types of data transmitted via the interface using the libraries of the 64 functions presented in the book [5]. System APROP was used to create software systems in air defense systems and VMF. The Assembly programming of systems from ready-made reuse and modules is created. It is protected in the doctoral dissertation [6] [7] [8]. The method of Assembly is also presented in the framework of generating programming K. Chernetsky and formulated models of transformation and configuration of applications by means of the language DSL (Domain Specific Language) and ADI (Architecture Description Language), etc. [9] [10] [11] [12].

The emergence of the OOP by G. Buch and UML has served as the impetus for the creation of the theory of object graph object and component modeling of complex systems based on logic and algebra-the mathematical theory of the description of the individual elements of complex systems and ensuring their variability according to the model of the basic characteristics of MF (Feature Model) as the mathematic apparat of the description of programs and systems [13] [14]. The development of the object paradigm is the theory of component programming and reusability Implementation and refinement of it spent Grishchenko V. N. in the system of "Informatization, NASU" and presented in his doctoral dissertation [15]. This theory is presented in the book [16] describes the component method and examples of building different component systems to ensure the reliability and quality of manufacturing such systems.

Due to the wide development of e-science and the spread of the Semantic Web and Web service tools, the ontological description of the domain of the life cycle of ISO/IEC 12207 standard is given and a formalized description of service and aspect programming from ready resources accumulated in repositories is given. Taking into account the theory of variability of systems from services and aspects, testing, configuring them into the system and evaluating the quality of the output files of the system [17] [18] [19] are proposed.

The author investigated the mechanisms of development of Internet Smart and Nanotechnology and formulated the General provisions of the development of smart computers and the concept of the transition of computer technology to nanotechnology [24].

2. The Fundamental Basis of System Programming

2.1. Basic Elements

The program is the object of development, which is run on the computer. Ready

program is a software product (PS) [9] [20]. Object design: module, program, system, family, etc.

A module is considered a software element that converts the plurality of source data X in a variety Y of the output method of display system. Modules are a pair $S = (T, \chi)$, where T—model of the system; χ is the characteristic function, defined on the set of vertices X of a graph of modules G.

The interface is the handler objects with each other to exchange data between them.

The development method is a method or systematic approach to achieve the goals which are set before creating the object of development. The method of modular programming is the decomposition of the problem into separate functions, each of which is a module and object, component, aspect, service and other methods.

Life cycle model PS—this is cascade, spiral, iterative, etc. On the basis of these models developed the first version of the standard ISO/IEC Life Cycle 1996, and then 2007. This standard give has a set of software development processes.

The technological process is an interrelated sequence of operations performed during the development of the object. The process is designed to transfer object from one state to another until final product [10].

Line technology (LT) and grocery PL specifies a set of development processes of some object, functions that convert to the ready program product (PP).

2.2. The Method of Programming Modular

The term module appeared in connection with the transition to the ES (IBM) computer (1976), which were implemented programming systems with languages ALGOL, FORNAN, PL/1, COBOL, and ASSEMBLER. Each language was independent of each other in form and content. Program in any of these languages got the name of the module. For the organization of communication of multilingual modules, an interface was formulated, which described the general data and data transfer operators from one module to another [9]. The method of modular programming was developed, which provides the formation of a modular graph structure, by which the Assembly of individual program elements from modules and interfaces was carried out.

The module is the basic software element with properties [16]:

-the logical completeness of function;

-the independence of one module from the other;

-replacement of individual module without disturbing the structure of the program;

-call other modules and return data to the call module.

The module converts the multiple *input* data X in a variety of *output* Y and is given as a mapping.

$M: X \rightarrow Y$.

Communication between modules:

-relationship management ($SR = K_1 + K_2$);

-connection according.

Modular graph structure $G = (X, Y)$, where

X is a set of vertices, and G is a finite subset of the direct product $X \times X \times Z$ on the set of arcs of the graph.

A modular structure is a pair $S = (T, \chi)$, where

T—the model of the modular structure;

χ—the characteristic function defined on the set of vertices X of a graph of modules graph G.

The value of the function χ is defined as:

$\chi(x) = 1$ if the module with vertex $x \in X$ included in the PS;

$\chi(x) = 0$ if module with top $x \in X$ is not included in the PS and it's not referenced from other modules.

Definition 1. Two models of modular structures $T_1 = (G_1, Y_1, F_1)$ and $T_2 = (G_2, Y_2, F_2)$ are identical, if $G_1 = G_2$, $Y_1 = Y_2$, $F_1 = F_2$. Model T_1 is isomorphic to T_2, if $G_1 = G_2$ between the sets Y_1 and Y_2, there exists an isomorphism φ, and for any $X \in X$, $F_2(x) = \varphi(f_1(x))$.

Definition 2. Two modular structures $S_1 = (T_1, \chi_1)$ and $S_2 = (T_2, \chi_2)$ are identical if $T_1 = T_2$, $\chi_1 = \chi_2$ and modular structures S_1 and S_2 are isomorphic if T_1 is isomorphic to T_2 and $\chi_1 = \chi_2$.

The module is described in PL and has a description section of the passport, which specifies external and internal parameters. To pass parameters to another module, use the Call (...), RMI (…) and another. The parameters may be converted to the form of the calling module and back in case of differences of their types. It was developed the library of primitive functions convert dissimilar data types PL [16]. This theory is suitable to the component.

2.3. Assembly Method

Assembly method based on the interface that connects the modules and data exchange. Interface first implemented in the system APROP (1975-1982) [9]. It can be inter-module, Interlingua and technological (1987) and became a fundamental concept in technologies programming SE [11] [20].

The *intermodule interface* is the contact modules to transmit and receive data between them. *Interlingua interface* is a library function interface to transform non-equivalent data types of the PL for IBM OS-360. Developed interface library (64 functions) which converts different data types (TD) in PL (ALGOL, COBOL, FORTRAN, PL/1, etc.). The system was handed over in 52 organizations of the USSR for Assembly of multi language modules to the applications in OS ES, IBM-360 (1982).

The formal conversion of objects TD Assembly is performed using algebraic systems Σ_1, Σ_2 for each TD PL T_α^t: $G_\alpha^t = \left\langle X_\alpha^t, \Omega_\alpha^t \right\rangle$,

where Ω_α^t —set of operations on the simple t (t = <*b-boolean*, *c-character*, *i-integer*, *r-real*> and complex t = <*a-array*, *p-pointer*, *u-union*, *s-sequence*> TD modern PL built classes of algebraic systems:

$$\Sigma_1 = \left\{\Omega_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r\right\},$$

$$\Sigma_2 = \left\{G_\alpha^a, G_\alpha^p, G_\alpha^u, G_\alpha^s\right\}.$$

Systems Σ_1 and Σ_2 the transformation TD $t \mapsto q$ for the pair of languages L_t and L_q have the properties:

1) G_α^t and G_β^q —isomorphic (to q and t defined on the same set);

2) X_α^t and X_β^q are isomorphic if Ω_α^t and Ω_β^q are different. If $\Omega = \Omega_\alpha^t \cap \Omega_\beta^q$ is not empty, then there is a isomorphism between $G_\alpha^{t\prime} = \left\langle X_\alpha^t, \Omega \right\rangle$ и $G_\beta^{q\prime} = \left\langle X_\beta^q, \Omega \right\rangle$.

3) Between the sets X_α^t and X_β^q may not be isomorphic matching, then build such a mapping between X_α^t and X_β^q that it is isomorphic.

If data types are different, for example, t—string, and type q is real, then there is no isomorphic correspondence between sets X_α^t и X_β^q. The maps preserve the linear order of the elements based on the linear order of the elements of the algebraic systems of these classes.

Theorem 1. Let φ—displays the algebraic system G_α^c to G_β^c. In order to φ be an isomorphism, it is necessary and sufficient to φ isomorphic reflected X_α^c and X_β^c, preserving linear order.

Assembly method and the library interface are also implemented in the complex PROMETEY by V. V. Lipaev and became the basis for the creation of quality software for different computers [11] [16]. As results, the Assembly programming was formed, which allows integrating heterogeneous software resources like reuses into complex systems by the method of configuration Assembly. For this method and participation in the creation of various complex programs, the author was awarded the prize of the Soviet of Ministers of the USSR (1985).

The Assembly was based on the theory of conversion of the fundamental TD (FDT) and later the common types of GDT. The FDT theory arose in the 70-ies of the last century in the works of Dijkstra, Hoare, Wirth, Ershov, Agafonov etc. Theory of general TD is defined in ISO/IEC 11,404-2006 (General Data Types), which allows the generation of the GDT↔FDT [21].

In 1990-1996 appeared languages of the description of interfaces—MIL (Model Interface Language), API (Application Program Interface) and IDL (Interface Definition Language). They are used in the configuration assemblies of dissimilar programs in modern PL (C, C++, Basic, Java, Python, etc.). Эта система была встроена в ОС ЕС (IBM-360). Later, there were development operators: *make* (BSD, GNU, Microsoft, Intel, UNIX and others), *building* in Grid technology, integration and so on. Operator make allows to collection executable files from software resources of libraries in the class of operating systems and programming systems. GNU Build System and Build Grid provide the Assembly of any software resources in different PL. The higher level of communication and interaction of program elements is provided by the standards OSI (1992) and WSDL (W3C.org).

Processing Data Types of PL

The data type TD is used in the description of PL programs (Algal, Prolog, PL/1, Fortran, Ada, Pascal, etc.). Axiomatic TD destroy developed by E. Dijkstra, N. Wirth, W. Tursky, P. Naur, A. Zamulin, N. Agafonov and others in the 1970. So on, which operate programs in PL include [9] [16] [21]:

-*FDT*—Fundamental Data Type implemented using primitive functions transform TD one LP to another and displayed in IBM-360 (1982);

-*GDT*—General Data Types (ISO/IEC 11404 GDT) for modern object-type PL (Basic, Java, Python, etc.);

-Big Data for Cloud Computing.

FDT and GDT DT—simple and complex.

Simple TD—integer (*i*), valid—real (*r*), boolean (*b*) and characterr (*c*) have a common form:

type $T = X\left(x_1, x_2, \cdots, x_n\right)$,

where T—the type name, $X\left(x_1, x_2, \cdots, x_n\right)$ names of values from the set of values of type T of X.

Operations (<, ≤, >, ≥, =, ≠) or (≤) determine the linear order of the elements of the set X. Operations on binary types include (true, false) and unary operations pred and succ, which define the previous and subsequent elements of the set X.

Complex TD—arrays, records, sets, union, etc. The array M shows the set of indices in the set of values.

M: $I \to Y$ and has the form of type T^{α} = array $T(I)$ of $T\left(\overline{Y}\right)$ задается в виде: $G^{\alpha} = <X^{\alpha}, \Omega^{\alpha}>$,

$$X^{\alpha} = \left\{x \middle| \left(\forall x_1 \in X^{\alpha}\right) \& \left(\forall x_2 \in X^{\alpha}\right) \Rightarrow I\left(x_1\right) = I\left(x_2\right) \right.$$
$$\left. \& \left(Y\left(x_1\right) \cup Y\left(x_2\right) \subset Y\left(X^{\alpha}\right)\right)\right\}, \Omega^{\alpha} = \{\leq\}.$$

In GDT TD uses simple and complex *TD*, based on the concept of cardinality and can be finite, accurate, infinite and approximate. A *Datatype generator* is an operation on one or more TD to create a new TD as a set of criteria for the characteristics of the TD using; procedures and a set of operations in the final value space to define a new TD. *Aggregate datatype* is a generated TD, each value of which includes the names of the TD elements from the aggregate TD value space.

Operations of *TD generation* are carried out by means of operations with:

1) Zero arity to generate the values of this TD;

2) Unary operation (arity 1), which turns the TD value into a new value of the same TD or boolean value;

3) Arity 2 that transform the pair of values of the TD in TD of the same value or a boolean.

4) *N*-Arity, which converts an ordered *n*-group and TD values to a parametric type, or aggregate.

In the value space there is an order relation (order), which is given by the sign (≤) and satisfies the rules:

1) For each pair of values (a, b) from the value space the following condition is met $a \le b$ or $b \le a$ or both;

2) For any two values (a, b), if $a \le b$ and $b \le a$, then $a = b$;

3) For any three values (a, b, c), if $a \le b$ and $b \le c$, then $a \le c$.

The GDT <–> FDT conversion system are based on these operations, implemented in [20] and displayed in Net. It presents value type and reference type. Types-values are static types in STS (Common Types Systems); their values can take up memory from 8 to 128 bytes. Reference types use pointers to the objects they type, as well as mechanisms for storing in class library FCl (Framework Class Library) and releasing memory. Reference types include: object type, interface type, pointer type, etc. In 1990 IK NASU is commissioned by the State USSR the processor for Fortran, Pl/1 and Assembler for ES OS that handles these languages based on the language syntax setting in tabular form. The build module calls other components of the translator to the programs of machine view (H. M. Mishchenko. "About Assembly programming of language processors"—Intellectualization of information and computing systems—K: IK NASU, 1990).

3. Disciplines of SE

In connection with the 40-year anniversary SE (2008) the author proposed a classification of scientific disciplines in SE [16] [22]. Proposed discipline used in the program Curicula-13. Let us consider briefly their scientific basic.

Scientific discipline SE includes classic Sciences (theory of algorithms, set theory, proof theory, mathematical logic, discrete mathematics); theory of programming of the theory of abstract data, management science, etc. This discipline defines the basic concepts the objects and the formalism of the description of the system components and data description [23] etc.

Engineering discipline SE includes methods of using technology rules and procedures, processes, life cycle, methods of measuring and assessing the quality of development *PP*. This discipline defines the set of engineering methods, techniques, tools and standards focused on the production of the target PP. Basic concepts of engineering SE include: core knowledge SWEBOK; the basic process SE; infrastructure environment.

Discipline management SE is based on the theory of management of projects and the IEEE Std.1490 PMBOK (Project Management Body of Knowledge); method CRM (Critical Path Method) for the graphic representation of works, operations and their execution time; method of network planning PERT (Program Evaluation and Review Technique), etc. In the PMBOK defined processes lifecycle of the project and the main areas of knowledge and processes of planning, monitoring, management and completion.

Economic discipline SE. This discipline provides for the calculation of the different parties activities of developers in the implementation of the project and identify the costs, time and economic indicators according to the requirements of *PP*. Used methods: predicting the size of *PP* (FPA—Function Points Analyses,

Feature Points, Mark II Function Points, 3D Function Points, etc.); the evaluation effort for the development of *PP* by using models COCOMO and systems (Angel, Slim, Seer-SEM, etc.), as well as the quality of *PP*.

Production discipline SE determines the production of *PP* and makes a profit. In the area of SE mass produced products created by the famous firms Microsoft, IBM, Intel, and the factory programs, as well as the results of outsourcing (upgrading a legacy inherited) bring on large profits. The production of *PP* is based on the technological processes of manufacture of certain product types using the theory of the design and usage of tool environments [16].

4. Programming Paradigm

Paradigm (from Greek παράδειγμα, "example, model, pattern")—a set of fundamental scientific attitudes, the concepts and terms adopted and shared by the scientific community. Provider continuity of development of science and scientific creativity Thomas Kuhn called paradigms established systems of scientific views, in which research and development [25] [26].

In SE emerged programming paradigm. It is a set of ideas, concepts, theories and methods that determine the style of formal presentation of computer programs. This term R. M. Floyd defined in his work "The Paradigms of Programming" (Communications of the ACM. 1969. Vol. 22 (8) pp. 455-460), E. Dijkstra in the book "Discipline of programming" (M.: MIR, 1976) and D. Gris in the book "Science of programming", K. Holsted "Science programs", 1984. They defined it as a method of conceptualization and formal definition of programs and systems. Some parts of the theory are implemented in the framework of applied (functional, logical, automatic, etc.), theoretical (VDM, OOP, Z, B, OCM, FODA, etc.), system (parallel, distributed, etc.) and commercial (Agile, SCRUM, EX, etc.) programming [9] [16].

4.1. Scientific Foundations of Programming Paradigms

To introduce several programming paradigms developed formal apparatus in theoretical and applied design of individual software resources for building (configuration) in the software system. In [16] [25] describes the theoretical foundations of the paradigm of object, component, service, aspect and generating programming.

Object design theory is built with the use of base notions of formal specification, set theory and class theory of G. Booch, G. Frege triangle and the object model (OM) CORBA, utilizing the following principles:

-all essences of the domain are objects;

-each object is a unique element;

-all objects are determined at a certain abstraction level and are ordered according to their relations;

-object interoperability with the interfaces.

An object is singled out using object analysis with mathematical terms for de-

scription and clarification of object methods in the OOP being created.

According to G. Booch, "object-oriented approach = objects + inheritance, polymorphism, encapsulation"; OM also encompasses object classes and their relations (aggregation, associations, specializations, instantiation so on), as well as their behavior.

Object is a named part of actual reality with a certain abstraction level; a notion structure according to Frege triangle (denotation, sign, and concept).

Each object O_i belongs to the set of objects $O = (O_1, O_2, \cdots, O_n)$,

where $O_i = O_i$ ($Name_i$, Den_i, Con_i), $Name_i$ is a sign, Den_i is a denotation, Con_i is an object concept, $Con_i = (P_{i1}, P_{i2}, \cdots, P_{is})$ is determined upon a set of predicates P_{il} [25].

***Axiom* 1.** The subject domain designed with objects is an object itself.

***Axiom* 2.** The subject domain being designed may be an object within another domain.

When designing the domain, each object gets at least one property or description, semantics allowing its unique authentication among the set of all objects and to the set of predicates of properties and relations between objects.

Object property is defined on the set of objects belonging to the domain with the unary predicate with return value depending on its external and internal properties. Description is an aggregate of properties (in form of predicates) subjected to the condition of acceptance of truth value by no more than one predicate from these descriptions. *Relation* is a binary predicate that returns truth on each pair of objects in the set. The basic types of mutual relations are as follows:

1) set—set;

2) element of a set—element of a set;

3) element of a set—set;

4) set—element of a set.

These relation types correspond to operations: *generalization*, *specialization*, *aggregation*, *association*, *classification and instantiation*. Types relations 3), 4) are *subsumption*, *relation* (IS-A) and part-whole relation (PART-OF), respectively.

The implementation of the object paradigm is described below. Other paradigms are discussed in [11] [12]. In this paradigm elements of software resources are described in PL, and their interfaces in standard WSDL. The proposed formal apparatus of the object-component method (COM) are programming paradigm. It formalizes the resource Assembly into complex programs and systems using the method of Assembly programming. The OCM provides a mechanism logic-mathematic modeling of graph model object (OM) and FM (Model Feature). On these models made configuration resources into system or family systems.

4.2. The Paradigm of Object Programming

Object Model (OM) of the subject area (SD) is modeling on four levels [16] [25]:

- *Generalizing* for determining OM SD base notions without considering of their essence and properties;
- *Structuring* for ordering objects in the OM taking into account relationships between them;
- *Characterization* for forming concepts of objects on the base of them properties and descriptions;
- *Behavioral level* for descriptions of conduct depending on events (such as time).

In accordance with the **generalizing** level an object is considered a mathematical notion, as a class from the point of view of von Neumann-Betrays-Gödel set theory: $O=\left(O_0,O_1,O_2,\cdots,O_n\right)$, with O_0 being an object in the subject domain. A set of base functions is formed at this level, related to decomposition or composition changes to object denotations and concepts, performed by increasing or decreasing object quantity, as well as expansion or narrowing object concepts. These changes are subjected to the set rules and terms that ensure correctness of function implementation.

For the set O the object relations hold:

$$\forall i \quad (i>0)\Rightarrow\left(O_i\in O_0\right), \tag{1}$$

The **structural** level defines such notions as class, class instance, abstract class, etc. The set of objects is ordered and each of objects can be presented as a set or an element of a certain set.

That is, expression (1) is transformed into

$$\forall i>0 \ \ \exists j\geq 0 \quad (i\neq j)\wedge\left(O_i\in O_j\right), \tag{2}$$

The objects are located at the vertices of the graph. The main vertex corresponds to the name of the set O. At the next level there are members of this set.

In accordance with the **characteristic** level, for each of objects a corresponding concept is formed. If $O'=\left(O_1,O_2,\cdots,O_n\right)$ is a set of objects SD and $P'=\left(P_1,P_2,\cdots,P_r\right)$ is a set of unary predicates related to properties of SD objects. Concept of the object O_i is a set of assertions, built on the basis of predicates from P' that are true for the object. That is, the concept $Con_i=\{P_{ik}\}$, if a condition $P_k(O_i)=true$, where P_{ik} is the assertion for the object O_i according to the predicate P_k. Following these rules, the properties of objects are determined with the subsumption relation. Expression $A=(O', P')$ determines the algebra system of object concepts O' and predicates P' with operations:

- 0-ary operations that correspond to constants;
- Unary operations that correspond to the properties of objects;
- Binary operations that correspond to intercommunications between pairs of objects.

Predicates must meet specific conditions:

- Number of predicates suffices the conceptual design of the subject domain using its objects;
- Each predicate, its type and signature meets the essence of the corresponding

object.

Axiom 1. Every object of the SD process has at least one feature or property, which equates to the set of objects.

In obedience to the **behavioral level**, a sequence of object states and processes is determined in order to reflect transitions between states. Intercommunications between objects are formed on the basis of binary predicates, which are related to the properties of SD objects, and are detailed to implement interoperability between states of objects.

According to the concept described above, *class* is an object that reflects a certain set; instance of the *class* is an object belonging to a certain set, which is a class; *joint class* is a set equal to the direct sum of several other sets; crossbred class is a common part of several other sets; *aggregated class* is a subset of the cross product of several other sets. If an object is an element of another object, it is determined by the set. However, not every object is necessarily an element of another class. For example, an object corresponding to the entire SD in the OM is not an element of any other object in the model. Definition of objects is formulated under the condition: each object is a set or an element of a certain set. Object ordering is performed taking into account affiliation by using sets of natural numbers.

Algebra for object-oriented analysis of the SD is

$$\Sigma = \left(O', I', A', P'\right), \tag{3}$$

where $O' = \left(O_1, O_2, \cdots, O_n\right)$ is a set of objects; $I = \left(I_1, I_2, \cdots, I_n\right)$ is a set of interfaces for O; $A' = \left(A_1, A_2, \cdots, A_n\right)$ is a set of operations on elements of a set O; $P = \left(P_1, P_2, \cdots, P_r\right)$ is a set of predicates that determine properties of object concepts. Each of operations in A' possesses certain priority and arity, and also related to the corresponding acceptable descriptions of object concepts and operations from the set

A' = {decds, decdn, comds, comdn, conexp, connar}. That is, decds, decdn are decomposition operations, comds, comdn are compositions, conexp, and conner is narrowing [20] [25].

Theorem 1. Set of operations A' for the algebra on the objects O' is a system of actions in relation to the functions of four-level object presentation of the OM.

Operations of object analysis are:

- Specification of object, as a *class*, *class instance*, etc.;
- Operations above essences: 0-unary, unary, *binary*;
- Interrelation of *generalization*, specialization, *aggregation*, *classification*, *instantiation*;
- Operations of object behavior together with communications between descriptions and the time of their existence in the OM.

The SD model may be represented by an object graph $G = \{O, I, R\}$, defined on the set of objects O, interfaces I and relations between objects R:

- Set of vertices O replicates one-to-one relationships between objects in the SD;

– Each vertex corresponds to at least one interface $I_k \in I$ and relationships from the set R according to certain rules.
– There exists at least one vertex with dual set-object status that reflects the entire domain.

The set of objects-functions O is related to implementation methods for objects in the SD, which communicate between themselves via interface objects from the set I. That is, vertices from G are objects of two types—functional objects O, and interface objects I (Figure 1).

Interface objects contain data description that is passed by RPC, RMI, ORB requests with optional operations of data transformation to the proper format of the environment containing the object implementation. The result of communication of a pair of objects in the graph (e.g., O'_5 and O'_7) is a new interface object with the description of input (*in*) and output (*out*) parameters of a request or a communication protocol.

Axiom 2. Graph G, complemented with interface objects, is well-organized structurally (bottom-up) with regard to the control of fullness, surplus and removal of duplicate elements.

Objects may have several interfaces that can inherit interfaces of other objects, providing they provide services for the entire set of output interfaces.

The set of objects and interfaces of the graph is reflected by general or individual properties and descriptions of the OM. Verification of properties of objects are provided by the specific operations (classification, specialization, aggregation, etc.). Each operation is a pair wise comparison of the underlying object properties with their external characteristics. They are reliable in case the following condition holds: each underlying property is equivalent to the external property of object. If this condition does not hold, an element is removed from the set O and the graph.

Worked out formal mechanism of transition is from objects OM to components and interfaces of CM (component model). Component programming was defined by formal models (component, interface, component environment) and the operations of external and internal component algebra with facilities of transformation of unrelevantical TD.

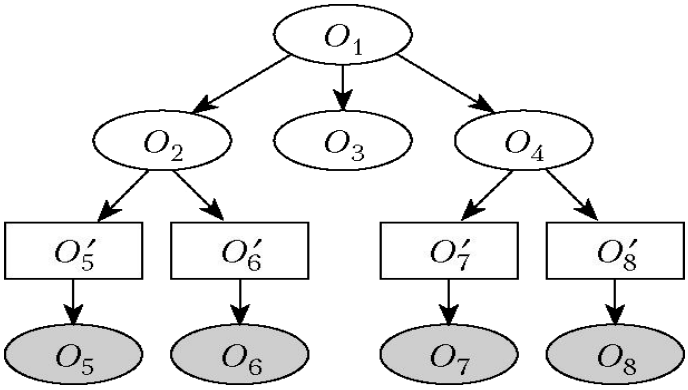

Figure 1. A graph G on the set of objects and interfaces.

4.3. The Paradigm of Component Programming

The concept of this paradigm was proposed in article [8] [26] and realized by V.N. Grishchenko in project of the informatisation NANY and was described in dissertation [15]. It was not defended it, he was died in 2009. This concept was continued in other works [26] [27] [28] [29]. On this paradigms design the component models (*CM*) of the system which include: model of component—*Comp*; interface model; model of the component environment *CE* and *CM*.

The CM obtained the formal elements [15] [25]:

CM = (*RC*, *In*, *Imp*, *Film*), where *RC*—the basic elements from the set components (*C*) and the OM; *In*—the interface of components with variation points; *ImC*—implementation of the base component in the environment; *Fim* (·)—functions for converting interface and data parameters in the interface signature. *OM* and *CM* are verified on the correctness of the inclusion of components *Comp*.

4.3.1. The Model of Component

MComp = (*CName*, *CIn*, *CFact*, *CImp*, *CServ*), where *CName* is the unique name of the component; *CIn* = {*CIni*}—the set of interfaces of the component; *CFact*—managing instance instances; *CImp* = {*CImj*} is the set of implementations of the component; *CSer* = {*CServ*}—a lot of system services. The set *CIn* = *CInI* ∪ *CInO*—input *CInI* and output *CInO* interfaces.

4.3.2. The Interface Model

CIn = (*InName*, *InFun*, *CIn*, *InSpec*), where *InName* is the name of the interface; *InFun*—interface functionality; *CIn*—interface for managing instances of the component; *InSpec*—specification of the interface (descriptions of types, constants, methods, etc.). The interface specifies the operation of managing instances:

Cln = {*Locate*, *Create*, *Remove*}, where

Locate—search and determine the instance of the component;

Create—create an instance of the component;

Remove—removes the instance of the component.

4.3.3. Model of the Component Environment

CE = (*Name Space*, *InRep*, *ImRep*, *CSer*, *CSerIm*), where

NameSpace—set of environment component names;

InRep = {$InRep_i$}—the repository of interfaces;

ImRep = {$ImRep_j$}—repository of implementations;

CSer = {$CSer_k$}—a set of system services;

CSerIm = {$CSerIm_i$}—a set of implementation of the services.

A component environment is a set of application servers where components, containers, and instances that implement component functions are deployed.

Thus, based on the CM model, the components of the *PS* can be distributed among the network nodes and interact with each other through the interface and

messages.

4.3.4. The Object and Component Algebra

$\sum = \{\varphi_1, \varphi_2, \varphi_3\}$, where

$\varphi_1 = \{CSet, CESet, \Omega_1\}$ is an exterior algebra;

$\varphi_2 = \{CSet, CESet, \Omega_2\}$ is an inner algebra;

$\varphi_3 = \{Set, CESet, \Omega_3\}$ is the evolution algebra;

$\varphi_4 = \{Set, CESet, \Omega_3\}$ is the assembling algebra.

Given algebras of operations on different elements of external, internal and evolutionary type provide transformation and Assembly and replacement of old with new ones.

4.3.5. Generating Programming (GP)

GP is a style of generating programs of different applications and their Assembly in the target system or family in two areas: applied engineering and engineering of the subject area [12]. The K. Chernetski model GDM (Generative Domain Model) is based on problem space, solution space and configuration base. The GDM model and the model of the characteristics of the MF indicate General concepts and characteristics of the elements of the domain, their interrelationships, as well as knowledge about the configuration of the system of reuse. For their implementation in multiparadigms of GP can use different programming styles: OOP, component, service, aspect, etc. In the production-based systems from ready made components in GP was taken the concept of product lines of the Institute SE USA (www.sei.com). For GP proposed the model transformation and configurations of system from ready-made components.

A transformation model is described in the DSL and contributes to the transformation of the space problems in the solution space by the method of transformation DSL-BOM component to their description in PL. The configuration model is based on design rules and domain abstractions included in GDM. Configuration model data is converted to concepts and their characteristics. The transformation of MF descriptions is performed in ADL language and then to the description of components in PL. The result is a configuration of system members as a configuration file.

4.3.6. Paradigms of Service Programming

The following types of services are different [11] [27]:

-General services of system environments (e.g. naming, cataloging, etc.);

-object services that manage objects, classes, and services (for example, request dispatching, interface management, etc.);

-network services of standard OSI model, SOA models (Service-oriented Architecture), SCA (Service-Component Architecture);

-ready resources (services, artifacts, reuses, assets, etc.);

-web-services Semantic-Web Internet.

These types of services are used in the modeling of information and software systems from ready services resources of the Internet. To use them, it is neces-

sary to search for a suitable Internet service resource, test it and embed it in the program for solving the problem in dynamic mode ([30] [32]).

In the CORBA system (1996) developed—object request broker (Object Request Broker—ORB) for communication between objects in different PL;

-general object services (Common Object Services—COS—facilities (Common Facilities—CF) providing (e.g., print, database, e-mail, etc.);

-the object of the application (Application Objects AO);

-client and server interface (stub, skeleton) via IDL.

Description of data transfer operations includes:

1) the name of the interface operation;

2) the list of parameters with arguments and parameters.

SOA (service-oriented architecture)—development of a component approach based on the use of services with standardized interfaces. They can be distributed across different network nodes. Their interface provides encapsulation of the implementation details of each component. SOA provides a flexible way to combine and reuse components to build complex distributed software systems and enterprise software applications [28]. The systems are implemented as a set of web-services integrated with standard Internet SOAP languages, WSDL W3C (www.c3.org).

Web-services are a new promising architecture that provides distribution at the Internet level [30]. Thanks to web-services, functions of any program in the network can be accessed via the Internet, and the results of accessing them—using PHP, ASP, JSP-scripts, JavaBeans, etc. An example of a web-service is the Passport system on Hotmail, which allows implementing user authentication. Web services are based on standards, open exchange protocols, data transfer in this order of action:

-definition of the format of requests to the web service and its responses;

-any computer on the network makes a request to the web-service;

-the web service processes the request, performs the action, and then sends the response.

The difference between web services and other technologies (e.g. named pipes, RMI) is that they are based on open standards and are supported on all UNIX and Windows platforms, etc.

A SOAP envelope contains a request to perform an action or a response. The envelope and its contents are encoded in XML and sent via HTPP to the web service. The problem with using web services is to find them. IBM, Microsoft have come up with an initiative project for Universal Description, Discovery and Integration (UDDI) to provide a common catalog of web services to enable all companies to "publish" their web service. The SOA (Service-oriented Architecture) model defines the system architecture from services, and the SCA (Service-Component Architecture) model specifies the architecture from service components.

These architectures are implemented by IBM Web Sphere Integration Devel-

oper. This system enables integration of SCA services through the JAVA interface model defined in the WSDL and JAVA interface class. J2EE sub modules and artifacts are packaged with an SCA module, whose function is to start the service and transfer data for integration [29].

Web services in the supports:

-SOAP Protocol to define the formats of the queries to network services (http://www.w3.org/TR/soap12-part1/wsdl20/RDF/wsci);

-WSDL—language service for.NET environment data exchange and Net Remoting network services development;

-UDDI (Universal Description, Discovery and Integration) to register, describe, store and search the registry;

-BPMN is a graphical notation language for application and business processes and BPEL—a language for describing business processes (http://www.omg.org/spec/BPMN).

The ready-to-use services in the application are configured in the output file for execution.

4.3.7. Aspect-Oriented Programming (AOP)

AOP is proposed in 1970. In the former USSR by A. L. Fuksman (Rostov University), as a technology of dispersed action or vertical lamination technology. Accordingly, this technology vertical layer (slice) contains a set of dispersed actions, code fragments that implement a certain extensible function, and the program development process is a set of operations to add and change these functions. Any program consists of modules designed for several horizontal layers. Any" extension "of the program results in a series of horizontal layer extensions [20].

G. Kishales and C.H. Simon companies Intentional Software continue to develop AOP. Each structural unit (component) of the system defines a procedure, function, or object, forming a functional element in the system. This element is localized and can be mapped to others. However, non-functional properties of the system, such as response to errors, providing access to memory (dynamic order-release), synchronization of parallel objects, etc., are usually "scattered" across all elements of the system, "was crossing" the structure of the system. Implementation of the properties of the aspect is carried out using:

-aspect language and linker (weaver) aspects in the application;

-connection points, determining the place in the program;

-slice—set of connection points according to the specified rule;

-fragment insertion—a set of instructions in the PL to integrate all points of the cut.

The aspect, by setting some slice or fragment, is to be inserted into the points of this slice.

Thus, the AOP offers different methods and techniques of splitting tasks (concern crosscutting) into a number of functional components, as well as aspects that "cross" functional components and provides for their composition in

order to obtain systems.

There are several implementations of AOP, for example, Xerox PARC AspectJ (www.aspectj.org) in the Java language. The release of Aspect J. 1.1 is built into the system, Eclipse, Sun ONE Studio and Borland J. Builder. IBM Research released a version of Hyper J. (www.alphaworks.ibm.com/tech/hyperj) and Cosmos (www.research.ibm.com/AEM/mdsoc.html) with hypertext support for building requirements and diagrams. Java AOP paradigm is the basis of C++, Squeak/ Smalltalk, Perl, Python, Ruby languages.

4.4. Processing Paradigm of Life Cycle Standard ISO/IEC 12207

The approach to automation ISO/IEC Life Cycle (LC) 12207-2007 is the ultimate tool serial process of manufacturing of *PS* using three categories of processes (**Figure 2**) [18] [19]:

1) The basic processes;

2) The support processes;

3) The organizational processes.

Development program starts with requirements, design elements of PS (modules, objects, components, etc.), integration (Assembly) of individual elements, testing of individual elements and overall system; operation ready PP. The supporting processes and organizational processes are used for quality management PS and software process improvement LC.

ISO/IEC LC 2007 (**Table 1**) includes 17 processes, subprocesses 74 and 232 tasks (actions).

These processes are necessary and sufficient for the design and manufacture

1. Category the "Basic processes"	**2. The support "Processes category"**
1.1 Order (agreement)	2.1 Documenting
1.1.1 Preparation of order, choice of supplier	2.2 Management by configuration
1.1.2 Monitoring supplier activity by user	2.3 Providing a quality guarantee
1.2 Delivery (acquisition)	2.4 Verification
1.3 Development	2.5 Validation
1.3.1 Exposure of requirements	2.6 General review
1.3.2 Analysis of system requirements	2.7 Audit
1.3.3 Planning system architecture	2.8 Decision of problems
1.3.4 Analysis of system requirements	2.9 Providing product applicability
1.3.5 Planning the system	2.10 Evaluation of product
1.3.6 Constructing (code) the system	**3. Category the "Organizational processes"**
1.3.7 Integration of the system	3.1 Management
1.3.8 Testing the system	3.1.1 Management at level organization
1.3.9 System integration	3.1.2 Management by project
1.3.10 System testing	3.1.3 Management by quality
1.3.11 Installation of the system	3.1.4 Management by the risk
1.4 Exploitation	3.1.5 Organizational providing
1.4.1 Functional application	3.1.6 Measuring
1.4.2 Support of user	3.1.7 Management by knowledge's
1.5 Accompaniment	3.2 Improvement
	3.2.1 Introduction of processes
	3.2.2 Evaluation of processes
	3.2.3 Improvement of processes

Figure 2. The basic, support and organizational processes LC.

Table 1. Process, subprocess and task of standard ISO/IEE 12207.

Class	Process	Action	Task
Basic processes	5	35	135
Support processes	8	25	70
Organisational processes	4	14	27
All	17	74	232

of any system. Some system companies sold individual pieces, i.e. individual variations of this standard or life cycle models (spiral, waterfall, iterative, etc.). The concept of automation of the Life Cycle of the ISO/IEC method of ontology is new and original. The basis of its implementation is the structure of processes of LC (**Figure 2**) and their interaction (**Table 1**), as well as the Ontology language for the conceptualization of individual variants of the process LC [18] [19].

The concept of automation of LC of the ISO/IEC 12207 by ontology is new and original. Formally, the LC model includes the processes P (process), actions (Action) and tasks T (Task):

$$M_{LC} = \left(P_k, A_m, T_n\right),$$

where $P_k = (P_{1k}, P_{2k1}, P_{3k2})$, $P_{1k} = 1 - 5$ (the main processes of the LC), P_{2k1}, $k_1 = 1 - 8$ are additional processes LC, P_{3k2}, $k_2 = 4$—organizational processes; $A_m = (A_{kr}, A_{k1l}, A_{k2j})$—actions or tasks of the process. In them the tasks mean:

A_{kr}, $r = 1 - 35$—actions on the main processes of the LC;

A_{kl}, $l = 1 - 25$—actions on the processes of support of the LC;

A_{k2j}, $j = 1 - 14$—actions in the organizational processes of the LC;

$T_n = (T_{nk}, T_{nl}, T_{nj})$ - T_{nk}, $\kappa = 1 - 135$—the tasks of the main processes LC;

T_{nl}, $l = 1 - 70$—the tasks of the support processes of the LC;

T_{pj}, $j = 1 - 27$—the tasks of the organizational processes of the LC.

An example of a description of the main processes of the LC (**Figure 3**) in XML language is given below.

```
<?xml version="1.0" encoding="utf-8"?>
<Association Line Name ="Define Requirements» Type="Main. Define requirements" Manually Routed
= "true" FixedFromPoint = "true" </AssociationLine>
<Association Line Name = "Integration PS" Type = "Main. Integration PS"
<AssociationLine Name = "Instlation" Type = "Main. "Instlation"
<Association Line Name = " An example of a description of the main processes of the LC (Fig. 2) in the XML language is given below. Exploitation" Type = "Main. An example of a description of the main processes of the LC (Fig. 2) in the XML language is given below. Exploitation"
<Property Name = " Integration PS" />
<Property Name = "Instlation" />
<Property Name = "Analysis Requirements" />
<Property Name = " Exploitation" />…
```

Figure 3. The description the main processes LC in the XML language.

Definition of processes of LC can be: the languages OWL (Web Ontology Language), ODSD (Ontology-Driven Software Development), XML (Extensible

Markup Language); systems modeling domain ODM (Organizational Domain Modeling), FODA (Feature-Oriented Domain Analysis), DSSA (Domain-Specific Software Architectures), DSL (Domain Specific Language), Eclipse DSL Tools vs. Net, Protégé, etc. this also includes the language of BPMN process description and life cycle the DSL to describe the semantics of the domains. Held ontological description of the main processes in the DSL The approach to the automation of the life cycle was presented at the conferences "Science and Information—2015", www.conference.thesai.org [19].

4.5. Method to the Creation of New Technologies

A technology for PP production amount of product lines and technologies. They are created using the method of technological preparation development (TPR, 1987) [10]. This method has been tested in the project of the Institute of Cybernetics AIS "Jupiter" for automation of the Navy of the USSR (1982-1991). It has developed six TL for creating and presents specific forms, documents and processes of these AIS. In this TL was sold about 500 of data processing programs for different objects AIS. TL processes perform the operations on the prepared resources (modules, components, data, etc.).

Work in the field of meta technologies TPR began to run through languages UML, DSL, Workflows, (BPMN Basic Process Modeling Notation), etc. These funds are used to create product lines (Product Lines/Product Family) as the infrastructure for the production of PP from ready resources, reuses [20] [21].

4.6. Factory Software-Based Industry Programs

Definition: Factory is an integrated architecture the Assembly line production of PP from ready-made software components (modules, objects, services, aspects, etc.), typically decorated in PL, and their interfaces in the WSDL. Last posted in system libraries and repositories [31] [32].

Analysis of the available factory programs (Grinfield, Bey, Lenz, etc.) and experience the creation of a specific student factory in KNU (http://programsfactory.univ.kiev.ua) allowed us to formulate the following set of necessary elements for the work of the factory programs:

1) prepared software resources (artifacts, modules, programs, systems, reuses, assets, etc.);

2) interfaces—qualifiers ready resources in one of the languages IDL, API, SIDL, WSDL;

3) TL, product line (Product Lines) production of PP;

4) the Assembly, Conveyor Line;

5) methods and techniques for the planning and execution of works on the line on creation of system;

6) system-wide development environment for individual programs.

On such method to do the existing factories programs:

1) AppFab in the system of collective development VS.Net;

2) AppFab IBM to create business systems;

3) AppFab in the CORBA system for the Assembly of heterogeneous software resources;

4) Product Line SEI USA;

5) Factory streaming building software John. Grinfeld, G. Lenz, I. Bay etc.;

6) Factory continuous integration by M. Fowler; etc.

Some factories are represented on the website http://www.7dragons.ru/.

5. Modeling Variability Systems with Paradigms

5.1. The Essence of Modeling of Systems and Families

The concept of variability of the original represented in the model FM (Model Feature) to Product Line based on the set of components reuse (CRR, Reuses), which in PS may include the variant points [11] [20].

Variability is a property of the system to the extension, modification, adaptation, or configuration for use in a particular context and to ensure its subsequent evolution (ISO/IEC FDIS 24745-2009 E).

Model FM is formed in the process of development of the PP and includes general functional and non-functional characteristics of items that can be used by family (FPS) members of PS when you create different variants of *PS* or *PP* on the points of variance.

The point of variance is a place in the system, which is used for the selection of the PS option. This point is a collection of options attached to the kernel made the system. In the production of the PS from CRU is created and the family PS. The FM model is used in engineering subject field and engineering applications for Assembly made resources.

Domain engineering provides the definition and implementation of common artifacts-variable functions for the production of a new product variant.

Artifacts—the architecture, requirements, components, tests, etc.

Application engineering includes the definition of artifacts needed by the user, and makes changes to the collection at the application level.

Theorem: The functional interaction between the two objects is correct, if the first object completely provides the functions and data transfer that is required by another object: $In(fo_k) \subseteq Out(fo_l)$.

The objects of the graph G (**Figure 2**) form a model of the system under system configuration. Elemental which can be changed in the graph are labeled by points of the variance (variability) [10] [11] [22].

The *point of variance is* in one place in the model system PS, which selects the variant of the system. A point of variance is handled by the configurator and allows transforming the prepared system by replacing some of the components used reuse, *CRR* by other more functional or correct.

Variability—a property of a product (system) to expand, change, adaptation, or configuration for use in a particular context and ensure its subsequent evolution ISO/IEC FDIS 24765-2009 (E).

5.2. Managing Variability of Systems

Model of variability PS – $MF_{var} = (SV, AV)$, where

SV—submodel of variability of the artifacts in the structure of *PS*;

AV—submodel variability of the finished product *PS*.

The MF_{var} model ensures that the artifacts of the *PS*, lower costs and decrease the cost of developing the system. Model of variability of *PS*—a set of *FM* models of the *PS*, set on the many artifacts, some points of variance for subsequent changes individual elements [20] [25].

Managing variability PS is performed on the points of variance, variant artifacts of the *PS*, limitations and dependencies by using the predicates *P* defined on the set of options of *PS*.

To control the variability method is used *E*. Deming, based on the functions F_1 - F_4:

F_1—operation, action to ensure that the artefacts of the *PS* (**Act**);

F_2—the planning system *FPS* of the artifacts (**Plan**) for engineering subject area and engineering applications;

F_3—system monitoring and verification of state changes of the *PS* (**Check**);

F_4—actualization (fulfillment) systems *PS* (**Do**).

Managing variability the *PS* with the requirements—*R* is:

1) the rationale for the function F_1 (R_1);

2) coordination of the implementation of artifacts in the processes of *PS* (R_2);

3) implementation of the validation of the creation of *PS* (R_3);

4) tracking relationships between the characteristics of the *PS* (R_4).

Compliance requirements R_1 - R_4 functions F_1 - F_4 model of environment process model process variability of the SPS is the basis for the formation and implementation of various systems [21].

The configuration model PS [21]:

$$M_{konf} = \left(OM, M_{SD}, M_{ps}, MF_{var}, M_{in}\right),$$

where

M_{in}—a model of interaction of individual elements of the created system.

Based on the model M_{konf} are:

-slection of artifacts and resources of the *PS* in the base configuration of a given system;

-allocation of common and variant characteristics of the *PS* in the model *FM* and model of the *PS*;

-planning for multiple resource use for PS in the points of variability and their fixation for their removal replacement;

-build resources in the *PS* and their adaptation to new conditions of environment;

-management options for *PS* with the replacement of individual functions in the *PS*;

-manage the interaction of artifacts in a heterogeneous environment.

5.3. Verification and Testing of the PS

For verification purposes the objects of systems use temporal logic (Linear Temporal Logic (LTL) or a logic tree computation CTL (Computational Tree Logic) [20] [25].

The method of deductive analysis LTL provides a logical output according to the model, made by hand. It applies only to those facilities that are critical (e.g. security of operation, or the protection of information).

Verification by model checking is only applicable to objects with a finite number of States. The feature of the method of verification for the model is that the verification is conducted automatically and do not need special knowledge and time. The method of verification—mathematical formulation of requirements to create programs with help algorithms of formal verification requirements.

Testing work products (plans, test suites, test data) is based on the use of CRU and finished products. Test products should be suitable for other PP and are part of the reusable components of a family of *FPS*. For testing the *PS* and *FPS* requirements use scenarios (Scenario-based test derivation), the method of analysis of trees FCTA (Fault Contribution Tree Analysis) and complex PLUTO (Product Lines Use case Test Optimization).

5.4. Evaluation of the Quality and Reliability of PS

The quality—the totality of the properties of *PS* that provide the ability to meet established or anticipated needs, in accordance with a purpose. The key characteristics of quality attributes are reliability and completeness as properties of the PS to eliminate failures with hidden defects with this criterion and a quality model, which relates the measures and metrics of the internal, external and operational type. From the standpoint of completeness of the product is the main indicator of quality are defects and failures [31] [32].

The main indicators of quality are defects and failures. This correspond such model of quality M_{qua}:

1) internal measure D_0 is the number of defects in each object *PS*;

2) external measure $R(t)$ is reliability of operation of each object in *PS* for a given time t without failure;

3) measure performance Q_{ps} is determined by the trouble-free functioning of the *PS*.

The model of defects based on multiple quality factors, analysis of causal relationships between them, combining qualitative and quantitative assessments of their impact on the density of defects. To calculate the *reliability function* uses a special formula:

$$R(t \mid T) = \exp\left(-\left(m(T+t) - m(T)\right)\right),$$

where t—the operating time of *PS* without a failure when testing in a period of time T;

$m(T)$ is a function of reliability growth, as the average number of defects *PP* identified during its operation for time *t*.

The reliability of the software largely depends on the number remaining and corrected errors in the development process. During operation, errors are also detected and eliminated. If the bug fixes are not made new, or at least new bugs introduced is less than clear, in the course of operation reliability increases.

The function of reliability growth $m(t)$ is defined by the formula

$$m(t) = N_0\left(1 - \exp\left(-\frac{\lambda_0}{N_0}\cdot t\right)\right),$$

where

N_0—the number of latent defects in *PP* at the beginning of system testing on form:

$$\lambda_0 = N_0 \cdot \frac{\rho \cdot K}{I \cdot \varphi}$$

λ_0—the failure rate of *PP* at the beginning of system testing, as defined by a given formula;

ρ—intensity code execution (speed of processor);

$K = 10^{-7}$—the ratio of defects (permanent) for model J. Musa;

I—number of source code instructions;

φ—code expansion ratio (the number of code instructions executed per original instructions).

To assess the *quality* systems used the standard ISO/IEC 9000 (1-4) quality model is form:

$M_{qua} = \{Q, A, M, W\}$, where

$Q = \{q_1, q_2, \cdots, q_i\}, i = 1, \cdots, 6$,—various quality characteristics (Quality—Q);

$A = \{a_1, a_2, \cdots, a_j\}, j = 1, \cdots, J$,—the set of attributes (Attributes—A), each of which captures a separate property of the qi quality characteristics;

$M = \{m_1, m_2, \cdots, m_k\}, k = 1, \cdots, K$,—the set of metrics (Metrics—M) each element of the attribute a_j for the measurement of this attribute.

$W = \{w_1, w_2, \cdots, w_n\}, n = 1, \cdots, N$ are weight coefficients (Weights—W) for metrics of the set M.

The quality standard identifies six basic quality characteristics: q_1: functionality; q_2: reliability; q_3: use, q_4: efficiency; q_5: maintainable; q_6: portability. The quality q_1 - q_6 are assessed by the formula:

$$q_1 = \sum_{j=1}^{6} a_{1j} m_{1j} w_{1j}$$

On the basis of obtained quantitative characteristics of the final grade is calculated by summing the values of individual indicators and their comparison with the benchmark systems.

5.5. CASE—Instrumental Tools

As a means of realization of life cycle processes ISO/IEC 2007 elected the lan-

guage Device and the DSL

Tool VS.Net etc. In them ontological description transformer to the XML language, it is the implementation language of the marked features of the LC domains, which define the communication and data exchange between them. *OCM* realized on the website http://7dragons.ru/ru. Site database forms a repository of ready resources, models PS, variability and interaction of system-wide tools—Visual Studio, Eclipse, CORBA, WSphere [20] [33]:

1) Visual Studio. Net↔Eclipse defines the environment of the interaction of individual elements in the C# language and interface. The model establishes the relationship of the elements with a given environment via the config file.

2) CORBA↔JAVA↔MS. Net provides communication between these environments with specified in these languages, the elements to access them from other developers.

3) IBM Sphere↔Eclipse provides communication between programs in *PL* these environments.

With the participation of the students was developed a program of processing *FDT* and *GDT*, a variant of the ontology LC using the tools, DSL Tools vs. Net and Protégé [16] etc. They are available on the website.

It is a link to a website of programs of KNU
http://programsfactory.univ.kiev.ua It accumulates the scientific artifacts of the students of KNU. The website also includes courses in Java, C # vs. Net and "Software engineering" for students of KNU and MIPT.

6. The Futute Technologies Internet and Nanotechnologies

6.1. The Future Internet Technologies [24]

1) The information objects (IO) that specifies the digital projection of real or abstract objects that use Semantic Web Ontology interoperability interfaces. IO through Web services began more than 10 years ago. Interaction semantics IO is based on RDF and OWL language of ISO 15926 Internet 3.0. The next step of the development of the Internet is Web 4.0, which allows network participants to communicate, using intelligent agents.

2) A new stage in the development of enterprise solutions-cloud (PaaS, SaaS) who spliced with Internet space and used to create Adaptive applications. Cloud services interact through the Web page by using agents.

3) Internet stuff (Internet of Things, Smart IoT) indicates the Smart support competing APPS using distributed micro services such as Hyper cat (mobile communications); industrial Internet (Industrial), covering the new automation concepts-smart energy, transportation, appliances, industry, and another.

6.2. Concept of Nanotechnologies

The idea of an Assembly of atoms in macro atoms special programs assemblies proposed by R. Feynman (1959) in the form of a manipulator of an atom, which are gravity, and the action of intermolecular Vander-Valesov force [24]. Can be

an arbitrary number of such mechanisms (machines) submitted by the manipulator of the elements, reduced to four and more times the copies of the "hands" of the operator, which can tighten small bolts and nuts, drill a very small hole, to perform the work in scale 1:4, 1:8, 1:16.

Nanotechnology is the technology of production of new materials and devices with predetermined atomic architecture (E. Dressler).

An atom is 10^{-10} = 1 nanometer (nm), and the bacteria is 10^{-9} nm. Particles from 1 to 100 nanometers are called nanoparticles. Some nanoparticles have the property of sticking together with each other, which leads to the formation of new agglomerates (in medicine, ceramics, metallurgy, etc.).

One of the greatest challenges facing nanotechnology is how to make the molecules group in a certain way and to organize themselves so to finally get a new material, substance or device. For example, proteins can synthesize from few proteins of DNK in complex structures with specific new properties.

6.3. Computer Nanotechnology

Today computer nanotechnology is actually already working with the smallest elements, "atoms" similar to the thickness of the thread (transistors, chips, crystals, etc.). For example, a video card from 3.5 million particles on single crystal, multi-touch maps for retinal embedded in the eyeglasses, etc.

Computational geometry is a part of computer graphics and algebra. Used in the practice of computing and control machines, numerical control etc. is also used in robotics (motion planning and pattern recognition tasks), geographic IS (geometric search, route planning), design chips, etc.

In the future, ready-made software elements will be developed in the direction of nanotechnology by "reducing" to look even smaller particles with predetermined functionality. Automation of communication, synthesis of such particles will give a new small element, which will be used like a chip in a small device for use in medicine, genetics, physics, etc.

7. Conclusion

The scientific, theoretical bases of the systems programming from modules, objects, components, services were presented. Defined by the theory of programming different periods of building systems, developed by the author with the participation of her students and postgraduates: theory of graph modular programming with graph representation in the form of adjacency matrix for mathematical relation of graph vertex attainability; implementation of complex complexes and systems based on graph, in the vertices of which there are different language modules (1970-1991); theory of program factories (2009) of modules and reuses, which has a great demand in different countries (Europe, India, China, Taiwan, etc.); theory of graph object and component modeling (OCM) with the help of logic and algebra-mathematical theory of formation of separate variants of complex systems and ensuring the variability of such systems on the

basis of the model of basic characteristics of MF (Feature Model) functional elements of the simulated systems (1997-2016); development of the theory of programming systems based on ontological and service-component models (SOA, SCA) with security and quality systems. The author investigated the mechanisms of development of Internet Smart and Nanotechnology and formulated the General provisions of the development of smart computers and the concept of the transition of computer technology to nanotechnology.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Erchov, A.P. (1974) Introduction to the Theory of Programming. Science.

[2] Glushkov, V.M., Lavrishcheva, E.M., *et al*. (1975) APROP System. 136 p.

[3] Lavrischeva, E.M. (1978) Problems of Combining Different Language Modules in the ES OS. No. 1, Programming.

[4] Grishchenko, V.N. and Lavrishcheva, E.M. On Creation of Interlanguage Interface for ES OS. No. 1, USIM.

[5] Lavrischeva, E.M. and Grishchenko, V.N. (1982) Communication of Multi-Language Modules in ES OS. Moscow, 127 p.

[6] Lavrischeva, E.M. (1988) Methods, Tools and Instruments of Assembly Programming. 34 p.

[7] Lavrischeva, E.M. and Grishchenko V.N. (1991) Assembly Programming. 213 p.

[8] Lipaev V.V., Pozin, B.A. and Shtrik, A.A. (1992) Technology of Assembly Programming. 272 p.

[9] Lavrischeva, E.M. (2006) Programming Methods. Theory, Engineering, Practice. Nauk. Dumka, Kiev, 451 p.

[10] Lavrischeva, E.M. (1987) Basics of Staging of Development of the Applied Programs ODS. Academy of Sciences, Ukrainian, 30 p.

[11] Lavrischeva, E.M. (2016) Assemblling Paradigms of Programming in Software Engineering. *Journal of Software Engineering and Applications*, **9**, 296-317. https://doi.org/10.4236/jsea.2016.96021

[12] Lavrischeva, E. (2013) Generative and Composition Programming: Aspects of Developing Software System Families—Cybernetics and Systems Analysis. Vol. 49, Springer, Berlin, 110-123.

[13] Booch, G. (1998) Object-Oriented Analysis. Binom, 560 p.

[14] Berger, T., She, S., Lotufo, R., Wąsowski, A. and Czarnecki, K. (2013) A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, **39**, 1611-1640. https://doi.org/10.1109/TSE.2013.34

[15] Grishchenko, V.N. (2007) Theoretical and Practical Applications of Component-Oriented Programming. 34 p.

[16] Lavrischeva, E.M. and Grishchenko, V.N. (2009) Assembly Programming. Basics of Software Systems Production. 371 p.

[17] Lavrischeva, E., Stenyashin, A. and Kolesnyk, A. (2014) Object-Component Devel-

opment of Application and Systems, Theory and Practice. *Journal of Software Engineering and Applications*, **7**, 756-769. http://www.scirp.org/journal/jsea https://doi.org/10.4236/jsea.2014.79070

[18] Lavrischeva, E.M. (2015) Ontology of Domains. Ontological Description Software Engineering Domain—The Standard Life Cycle. *Journal of Software Engineering and Applications*, **8**, 324-338. http://www.scirp.org/journal/jsea https://doi.org/10.4236/jsea.2015.87033

[19] Lavrischeva, E. (2015) Ontological Approach to the Formal Specification of the Standard Life Cycle. *Science and Information Conference*—2015, London, UK, 28-30 July 2015, 965-972. http://saiconference.com/Conferences

[20] Lavrischeva, E.M. (2014) Software Engineering Computer Systems. Paradigms, Technologies, CASE-Tools Programming. Nauk. Dumka, Kiev, 282 p.

[21] Lavrischeva, E.M. and Ryzhov A.G. (2016) Application of the Theory of Common Data Types of ISO/IEC 12207 GDT to Big Data, "Actual Problems in Modern Science and Ways of Their Solution", 27 December. http://euroasia-science.ru

[22] Lavrischeva, E.M. (2008) Classification of Software Engineering Disciplines. *Cybernetics and Systems Analysis*, **44**, 791-796. https://doi.org/10.1007/s10559-008-9053-5

[23] Lavrischeva, E.M. (2015) Component Theory and a Collection of Technologies for Development of Industrial Application of Ready Resources. *Proceedings of the 4—Scientific Practical Conference* "*Actual Problems of System and Software Engineering*", *OPSPI* 2015, 20-21 May 2015, 101-119.

[24] Lavrischeva, E.M. and Petrov, I.B. (2017) Ways of Development Computer Technologies to Perspective Nano. *Future Technologies Conference* (*FTC*) 2017, Vancouver, Canada, 29-30 November 2017, 978-991.

[25] Lavrischeva, E.M. (2016) Theory of Object-Component Modeling Software Systems. ISP, 48 p. http://www.ispras.ru/

[26] Gorodnia, L.V. (2017) Paradigms Programming: Analysis and сравнение. SORAN, 239 p.

[27] Lavrischeva, E.M. and Slabospitska O.L. (2015) Technology for Changing Software Enginering and System Modeling. Proceedings of 12th Conference TAAPSD—2015, Kiev, 23-26 November 2015, 118-127.

[28] Lavrischeva, E.M., Karpov, L.E. and Tomilin, A.N. (2017) Approaches to the Representation of Scientific Knowledge in the Internet Science. Sat. *XIX All-Russian Scientific Conference* "*Scientific Service on the Internet*", Novorossiysk, 18-23 September 2017, 310-326.

[29] Lavrischeva, E.M. (2016) Software Engineering. Programming Theory, MIPT, Textbook, 48 p. and Programming Technology, MIPT, 52 p.

[30] Lavrischeva, E.M. (2016) Software Engineering. Basic Foundation of Software Engineering, MIPT, Textbook, 51 p.

[31] Lipaev, V.V. (1983) Software Quality, Finance and Statistics, 320 p.

[32] Andon, F.I., Koval, G.I., Korotun, T.M., *et al.* (2007) Foundation Engineering of Quality PS. Akademperiodica, 680 p.

[33] Ostrovsky, A.I. (2011) An Approach to the Interoperability of Software Environments JAVA and MS. Net. The Problems of Programming. No. 2, 37-44.