International Journal of Applied
and Natural Sciences (IJANS)
ISSN(P): 2319-4014; ISSN(E): 2319-4022
Vol. 7, Issue 5, Aug - Sep 2018; 15-32

International Academy of Science,
Engineering and Technology
Connecting Researchers; Nurturing Innovations

THE SCIENTIFIC BASIS OF SOFTWARE ENGINEERING

E. M. Lavrischeva

Professor, Moscow Institute of Physics and Technology, Chief Scientific of ISP RAS, Russia

ABSTRACT

Define the scientifical concepts and fundamental basis of Software Engineering (SE). The scientifically concepts are modules, objects, programs, systems and processes of design program systems. The fundamental basic of the SE are: assembly method of modules; disciplines SE (scientific, engineering, economic, management, etc.); paradigms of programming modules, objects, components, etc.; automatization of Life Cycle (ISO/IEC 12207); theory of technological and production lines, factory programs and App Fabric; new logical-mathematical theory of object-component modeling graph of object model (OM); verification of OM and MF (feature model) and testing ready elements of system and evaluation reliability and quality of systems product. Tool support of scientific basis was represented on the website http://7dragons.ru/ru.

KEYWORDS: *Science, Concepts, Formalism of Modules, Assembly Method, Object, Component, Interface: IDL, Logic-mathematical Theory, Life Cycle, Product Line, Model FM, Configuration, Verification, Testing, Reliability, Quality*

Article History

Received: *08 Jun 2017* | ***Revised:*** *19 Jul 2018* | ***Accepted:*** *01 Aug 2018*

1. INTRODUCTION

Since the advent of the SE (1968), many scholars and specialists of Computer Sciences began to create methods and tools for designing programs and systems to solve tasks from different fields of knowledge (mathematics, physics, biology, architecture, industry, etc.). The first book by B. Boehm [1] is devoted to the description techniques of design and quality evaluation. In [2, 3] describes the methodology of designing and evaluation of reliability and quality systems. In connection with the emergence of the OOP in [4, 5] presents a unified modeling language (UML, 1994) systems with use case precedents and processes RUP. In Software Engineering Body of Knowledge (www.swebok.com, 2001, 2004, and 2014) the definition of the subject - Software Engineering. It is a system of methods, means and disciplines for the planning, design, operation, and maintenance designed for industrial production. SE covers all aspects of creation from the beginning of the formulation of requirements up to the development, maintenance, and decommissioning. In [6-9] the basic concepts are formulated, the paradigm of the programming and Assembly of software ready elements in a system or family of systems. The following are a scientific and formal framework of production systems of different objects, their verification, and testing, and evaluation quality assembly systems [10, 11].

2. THE FUNDAMENTAL BASIS OF SE

2.1. Basic Concepts

The program is the object of development, which is run on the computer. A ready program is a software product (PS) [4, 5]. 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 the display system. Modules are a pair $S = (T, \chi)$, where T – a 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 achieving 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 paradigms programming.

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 an object from one state to another until the final product [10].

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

Tools (CASE-tool factory, a system-wide tool) is software or a methodical means to obtain the object in a PP.

2.2. Assembly Method

Assembly method based on the interface that connects the objects, modules and data exchange. Interface first implemented in the system APROP from 1975 to 1982 in [6-9]. It can be inter-module, interlingual and technological (1987) and became a fundamental concept in programming technologies and software engineering [6-10].

The *intermodal 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 and applications in OS ES (1982).

The formal conversion of type data objects of the Assembly is performed using algebraic systems for each data type T_α^t: $G_\alpha^t = <X_\alpha^t, \Omega_\alpha^t>$,

Where, Ω_α^t - set of operations on t TD. For simple $t = <b, c, t, r>$ and complex $t = <a, z, u, e>$ TD modern PL built classes of algebraic systems:

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

$$\Sigma_2 = \{G_\alpha^a, G_\alpha^z, G_\alpha^u, G_\alpha^e\}.$$

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

- G_α^t and G_β^q – isomorphic (to q and t defined on the same set);
- 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 $G_\alpha^{t'} = < X_\alpha^t , \Omega >$ и $G_\beta^{q'} = < X_\beta^q , \Omega >$.
- 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.

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 PROTVA (V. V. Lipaev) and became the basis for the creation of software for different computers MVK. These complexes have been awarded the State prize of the Cabinet of Ministers of the USSR (1985).

The Assembly was based on the theory of conversion of the fundamental data types (*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 data types is defined *GDT* in ISO/IEC 11404 - 2006 (General Data Types), which allows the generation of the *GDT*↔*FDT*.

In 1992-1996 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.).

The Theory of Modular of Programming

The module is the basic software element with properties [10]:

- 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 caller module.

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

M: X →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 x X x Z on the set of arcs of the graph.

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

T – a model of the modular structure; χ is 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 the 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 ∈ 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 (...). The parameters may be converted to the form of the calling module and back in case of differences of their types. It was developed in the library of primitive functions convert dissimilar data types *PL* [16]. This theory is suitable to the component. Object and component the theory of programming discussed below.

3. DISCIPLINES OF SUBJECT SE

In connection with the 40 year anniversary SE (2008), the author proposed a classification of scientific disciplines in SE the articles [13, 14, and 10]. Proposed discipline used in the program Curicula-13. Let us consider briefly their characteristics.

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 of the objects and the formalism of the description of the system components and data description [13, 14] 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 [1] 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 the manufacture of certain product types using the theory of the design and usage of tool environments [12].

4. PROGRAMMING PARADIGM

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 [10-14] describes the theoretical foundations of the paradigm of the 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, Frege triangle and 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 description 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.

An 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 $O= (O_1, O_2. \ldots, On)$,

where, $O_i = O_i (Na_i, Den_i, Con_i)$, Na_i is a sign, Den_i is a denotation, Con_i is an object concept, Con$i$ = $(P_{i1}, P_{i2},\ldots, P_{is})$ is determined upon a set of predicates P_{il} [7-9].

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.

The 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. A 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. The *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:

- Set – set;
- Element of a set – element of a set;
- Element of a set – set;
- Set – element of a set.

These relation types correspond to operations: *generalization, specialization, aggregation, association, classification, and instantiation.* Types of 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 [15, 16].

Made in these paradigms, elements of software resources are described in PL and their interfaces in standard WSDL. The proposed formal apparatus of the object component programming paradigm. It formalizes the resource Assembly into complex programs and systems using the method of Assembly programming. This method provides a mechanism of interaction between resources of these paradigms in the new system or family. The theoretical components include functional, agent, automaton, etc.

4.2. Technical Programming Paradigms SE

The Agile methodology is focused on the close collaboration of a team of developers and users. It is based on a waterfall model lifecycle incremental and rapid response to changing demands on *PP*. The team works according to the schedule and financing of the project.

eXtreme Programming (XP) implements the principle of "collective code ownership". It any member of the group can change not only your code but also code another programmer. Each module is supplied with the autonomous test (unit test) for regression testing of modules. Tests written by the programmers and they have the right to write tests for any module. Thus, most of the errors are corrected at the stage of encoding, or when you view the code, or by dynamic testing.

SCRUM is Agile methodology project management firm Advanced Development Methods, Inc., used in organizations (Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox and Hewlett - Packard etc.) are based on an iterative Life Cycle model with well-defined development process, including requirements analysis, design, programming, testing (http://agile.csc.ncsu.edu).

DSDM (Dynamic Systems Development Method) for rapid development of RAD (Rapid Application Development) prototyping and ASD (Adaptive Software Development) for extreme projects and is based on the theory of complex adaptive systems. It is a static development Cycle PS includes Planning — Design — Programming, and dynamic Cycle of Reflection — sharing — Learning. *FDD* (Feature Driven Development) methodology focused on functionality

(www.nebulon.com) and is a model-driven process.

4.3. Processing Paradigm of Life Cycle Standard ISO/IEEE 12207

The approach to automation ISO/IEC Life Cycle 12207-2007 is the ultimate tool serial process of manufacturing of *PS* using three categories of processes:

- The main processes (Figure 1);
- Support processes (Figure 2);
- Organizational processes (Figure 2).

In this standard the following list of processes, their implementation, and forms of representation results. Core processes – these are processes of development, operation, and maintenance of *PS* (Figure 2).

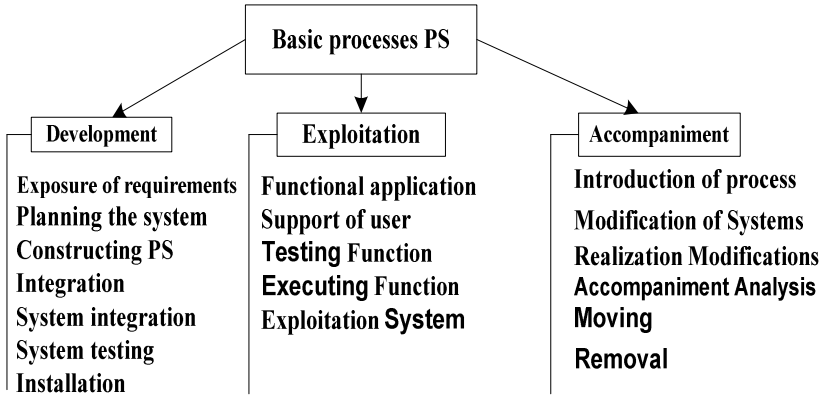

Figure 1: Diagram of the Main Processes of Life Cycle of the PS

Development process 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 (Figure 2) are used for quality management PS and software process improvement Life Cycle (LC).

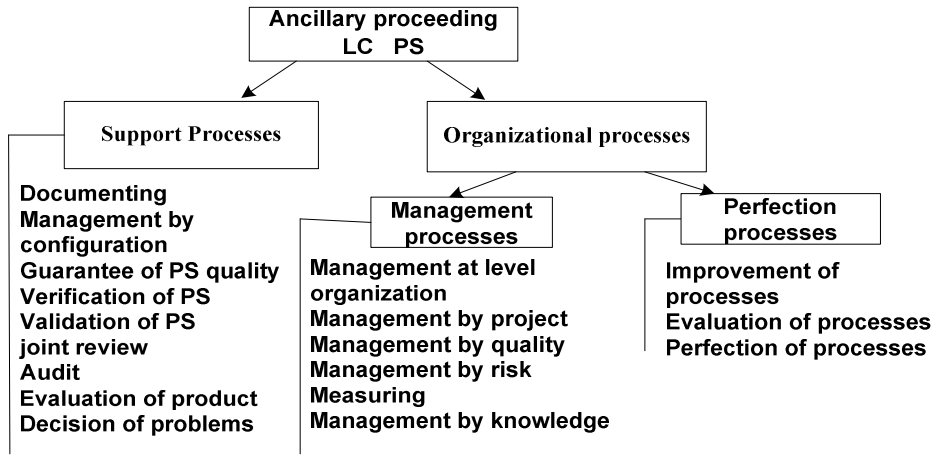

Figure 2: The Scheme Support Processes

ISO/IEC Life Cycle 2007 (Tabl. 1) includes 17 processes, sub-processes 74 and 232 tasks (actions).

Table 1: Process, Sub Process 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

These processes are necessary and sufficient for the design and manufacture 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 1, 2) and their interaction (table.1), as well as the Ontology language for the conceptualization of individual variants of the process LC [23, 24].

Definition of processes of life cycle 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 LC the DSL to describe the semantics of the domains. Held ontological description of the main processes, support processes and organizational processes in the DSL and the description of these processes in Protégé 2.3 in the XML format. The approach to the automation of the LC was presented at two conferences, including “Science and Information - 2015” in London [24].

4.4. Method to the Creation of New Technologies

An technology for PP production amount of product lines and technologies. They are created using the method of technological preparation development (TPD, 1987) [12]. This method has been tested in the project of the Institute of Cybernetics AIS "Jupiter–470" 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 TPD 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 and reuse [19, 20].

4.5. Factory Software – based Industry Programs

Definición

A 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 [17, 18].

Analysis of the available factory programs (Grinfeld, 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:

- Prepared software resources (artifacts, modules, programs, systems, reuses, assets, etc.);

- Interfaces - qualifiers ready resources in one of the languages IDL, API, SIDL, WSDL;
- TL, product line (Product Lines) production of *PP*;
- The Assembly, Conveyor Line;
- Methods and techniques for the planning and execution of works on the line on the creation of system;
- System-wide development environment for individual programs.

On such method to do the existing factories programs:

- AppFab in the system of collective development VS.Net;
- AppFab IBM to create business systems;
- AppFab in the CORBA system for the Assembly of heterogeneous software resources;
- Product Line SEI USA;
- Factory streaming building software John. Grinfeld, G. Lenz, etc;
- Factory continuous integration by M. Fowler; etc.

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

5. MODELING CHANGING SYSTEMS

5.1. The Essence of Modeling of Systems and Families

The concept of the variability of the original represented in the model FM (Model Feature) to Product Line based on the set of components reuse (CRU, Reuses), which in PS may include the variant points [19, 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.

5.2. New Method Modeling of Systems and Families

One of the new methods of modeling of systems is theory object-component method (OCM) [21, 22]. This method provides a four-level logical-mathematical design of families of FPS using the function ($Fo = fo_1,..., fo_n$) and interface elements ($Io = io_1,..., io_m$) domain.

Functional elements of a domain form a set $F_O = (f_{O1}, f_{O2}, ..., f_{On})$ and their *interfaces* – a lot of *Io*. Applies the unary predicates from the set $P' = (P_1, P_2,... P_r)$. These predicates establish the presence of the characteristic properties of the elements of the F_O. Functional object *foi* specifies a formal description of application functions *PS*, which provides the solution of the problem specified subject area/domain. The object is given by a triple: the name, data types, and their values. Front-end I_o object specifies the formal description of the operations call the methods and data of functional entities which are specified by the intermediary interactive functional objects.

Logical-mathematical modeling of *FPS* on the levels of functional and interface objects is reduced to the construction of sub graph of levels and to the final formation of the graph [21]:

G = (O, *I, R)*,

Where , *O* – the set of functional objects, *I* – the set of interface objects, *R* – the set of relations between objects (Figure 3).

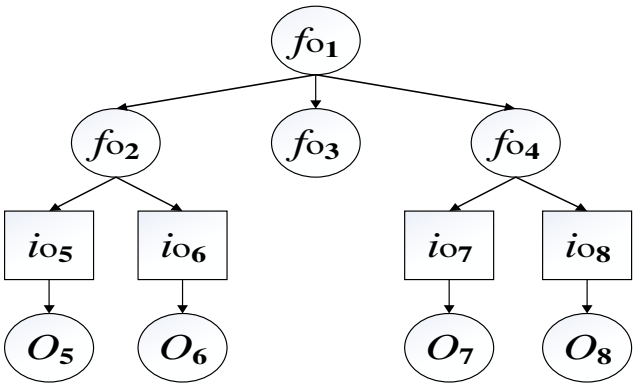

Figure 3: A Graph G on the Set of Functional and Interface Objects

The vertices of the graph G define the functional elements of the SPS – f_{O1}, f_{O2}, f_{O3}, f_{O4}, f_{O5}, f_{O6}, f_{O7}, f_{O8} and interface elements – i_{O5}, i_{O6}, i_{O7}, i_{O8}.

Functional elements of the graph $f_{O1} - f_{O8}$ described in a programming language, and interface objects i_{O5}–i_{O8} interface language IDL. The relationship between the functional objects, for example, O_k, O_l is provided by an interface object from the plurality of input interfaces *In*.

Graph *G* it is possible to collect individual programs $P_1 - P_6$ using mathematical operations $\cup$, the relevant activity link:

- $P_0 = (P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5)$;
- $P_1 = f_{O2} \cup f_{O5}$, *link* $P_1 = In\ i_{O5} (f_{O2} \cup f_{O5})$;
- $P_2 = f_{O2} \cup f_{O6}$, *link* $P_2 = In\ i_{O6} (f_{O2} \cup f_{O8})$;

- P_3 ;
- $P_4 = f_{O4} \cup f_{O7}$, *link* $P_4 = In\ i_{O7}\ (f_{O4} \cup f_{O7})$;
- $P_5 = f_{O4} \cup f_{O8}$, *link* $P_4 = In\ i_{O8}\ (f_{O4} \cup f_{O8})$.

These programs are part of substations and may include the variant points for change in the individual functional elements of the model *PS*, respectively, in the graph model, *G*. Presents the transition from objects to components. It defines the component model is adequate to the object in the code machine and a modified model of variability [25].

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* form a model of the system under system configuration. Elemental which can be changed in the graph is labeled by points of the variance (variability) [10, 11, 22].

The *point of the variance* 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, *CRU* 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).

5.3. The Transition from the OM to the Component Model (*MC*)

The essence of OCM is to provide domain count of *OM* and transformation of function objects to software components that are integrated, are configured in the *PS* components and interfaces and component operations of the algebra, and is isomorphic to contribute to display different types of data components associated with each other [17].

Component formally has the form:

$CIm_j = (ImNa_j, ImFunc_j, ImSpec_j)$, where

ImNaj – the name identifier of the implementation component;

$ImFunc_j$ - functionality appropriate for a given implementation (as a set of method implementations);

$ImSpec_j$ – implementation, specification (description of conditions, settings, etc.).

The model of the *component's interface* has the form:

$CIn_i = (InNa_i , InFun_i , CIn, InSpec_i)$, where

$InNa_i$ – the name of the interface;

$InFunc_i$ – functionality (methods) implemented by the given interface;

CIn_i – interface instance control component;

$InSpec_i$ – interface specification (the description of types, constants, other data members, method signatures, etc.).

Interface In_i from the formula interface model defines the conditions of management of class instances:

- Searching and identifying the required component – Locate;
- Create a component instance – Create;
- Removal of a component instance – Remove.
- Administration of components is CIn_i = {Locate, Create, Remove}.

For interaction between two components C_1 and C_2 are determined by the following necessary condition:

If, $CIni_1 \in CInO_1$, then there must be $CIn_2 \in CInI_2$ such that Sign ($CIni_1$)= Sign(CIn_2)&Provide($CIni_1$) $\subseteq CImj_2$ where Sign(.) means the signature of the appropriate interface.

Between object and component representations of programs, there is an ambiguity, which is generated by the fact that a particular component can have implementations for multiple interfaces Isyst.

If each of the interfaces implemented by a separate component, then there is a single equivalent mapping between object and component representations of the application structure.

To make changes in the CM model uses an component algebra - external, internal and evolutional algebras:

$\sum$ = {φ1, φ2, φ3}, where

φ1= {*CSet, CESet,* Ω1} – exterior algebra,

φ2 = {*CSet, CESet*, Ω2} – interior algebra,

φ3 = {*Set, CESet*, Ω3} is the algebra of evolution of components.

The set of operations this algebra is given in [25].

5.4. Managing Variability of Systems

Model of variability PS - MF_{var}= (*SV*, *AV*), where

SV – submodel of the 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 the variability of *FPS*- a set of *FM* models of the *PS*, set on the many artifacts, some points of variance for subsequent changes individual elements [22].

Managing variability FPS 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 artifacts of the *FPS* (**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 *FPS* (**Check**);

F_4 – actualization (fulfillment) systems *FPS* (**Do**).

Managing variability the *FPS* with the requirements - *R* is:

- The rationale for the function F_1 (R_1);
- Coordination of the implementation of artifacts in the processes of *FPS* (R_2);
- Implementation of the validation of the creation of *FPS* (R_3);
- Tracking relationships between the characteristics of the *PS* and *FPS* (R_4).

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

The configuration model *PS*, *–FPS* [21]:

$M_{konf} = (OM, M_{SD}, M_{ps}, MF_{var}, M_{in})$, where

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

Based on the model M_{konf} are:

- Selection 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.

5.4. Verification and Testing of the PS and SPS

For verification purposes, the objects of systems use temporal logic (Linear Temporal Logic (LTL) or a logic tree computation CTL (Computational Tree Logic) [10, 11].

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).

5.5. 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 is defects and failures [10].

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

- Internal measure D_0 is the number of defects in each object *PS*;
- External measure $R(t)$ – is the reliability of operation of each object in *PS* for a given time t without failure;
- 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:

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

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

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

N_0 – the number of latent defects in *PP* at the beginning of system testing on the 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 – a intensity code execution (speed of processor);

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

I – a number of source code instructions;

φ – a 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

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

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

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

$W = \{w_1, w_2,...,w_n\}$, n = 1,..., 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 the obtained quantitative characteristics of the final grade is calculated by summing the values of individual indicators and their comparison with the benchmark systems.

5.6. CASE –instrumental Tools

As a means of realization of life cycle processes, ISO/IEC 2007 elected the language Device and the DSL

Tool VS.Net etc. In them ontological description transformer to the XML language, which 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:

- 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.
- CORBA↔JAVA↔MS.Net provides communication between these environments with specified in these languages, the elements to access them from other developers.
- 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 [22] 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. The site was turned over 150,000 of the students and professors.

6. CONCLUSIONS

The paper presents the scientific concepts and the fundamentals of the subject of software engineering, presented in the monograph figure 4 [10] and its copy figure 5 [11]. Describes the fundamental scientific concepts SE (object, program, TL, tool, method, interface, etc.) and Assembly method based on interfaces. The described *OCM*,

including logical-mathematical apparatus of systems design to synthesis, structural, and behavioral characteristic levels. It is built graph of object model and model characteristics of the MF to control the variability of systems and their families *FPS*.

The formal theoretical description of programming paradigms– object, component, service, aspect and generating programming is suggested [10]. Show the standard descriptions of objects, interfaces, and Assembly method (configuration) for obtaining the modified complex systems. Present a new approach to creating complex systems with the use of *OM* graph and model *MF*. This is description of the idea of automation of processes of LC ISO/IEC 12207 by means of ontology [24] in the Protégé environment of the site the ISP http://7dragons.ru/ru.

7. REFERENCES

1. *Boehm B. W. Engineering design software. - M. Radio and communication. - 1986. - 510 p.*
2. *Lipaev V. V. the Reliability of the software of ACS, Energoizdat, 1981. (in Russian).*
3. *Lipaev V. V. Software Quality, Finance and statistics, 1983.-320p. (in Russian).*
4. *Jacobson Ivar. Object-Oriented Software Engineering: A Use Case Driven Approach, 1992.- ISBN 0- 201-54435-0.*
5. *Pfleeger Shari Lawrence, Software engineering: theory and practice, London : Prentice-Hall, - 1996.- 676 p.*
6. *Lavrischeva E. M. Grishchenko V. N. The connection of multi-language modules in OS, – M.: Finance and statistics, 1982.-137 pp. (in Russian).*
7. *Lavrischeva E.M. The problems of software engineering.-Knowledge. - K., 1991.-29c.*
8. *Lavrischeva E. M., Grishchenko V. N. The Assembly programming. – Kiev: Nauk. Dumka, 1991. – 213 c.*
9. *Lavrischeva E. M., Petrukhin V. A. Methods and means of software engineering security. – M.: MIPT- 2007. – 415 p. (in Russian).*
10. *Lavrischeva E. M. Software Engineering computer systems. Paradigm technology, CASE tools programming. K.: Nauk. Dumka.- 2014.-285 p. (in Russian).*
11. *Prakrit Trivedi, Anil Kumar Dubey & Vipul Sharma, Experimental Report to Analyse Human Thoughts on Software Engineering, International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR), Volume 2, Issue 4, November-December 2012, pp. 45-52*
12. *Lavrischeva E. M. Programming methods. Theory, Engineering, Practice.-K.: Nauk. dumka, 2006. -451 p. (in Russian).*
13. *Lavrischeva E. M. Basics of staging of development of the applied programs ODS.- Preprint 87-5 ICyb. Ukrainian, Academy of Sciences.-1987.-30 p, (in Russian)..*
14. *Software engineering as a scientific discipline and engineering //E. M. Lavrishcheva, 2008, Volume 44, Number 3, Pages 324-332.*
15. *Classification of software engineering disciplines. E. M. Lavrischeva 2008, Volume 44, Number 6, Pages 791-796.*

16. *Ekaterina M. Lavrischeva. Assemblling Paradigms of Programming in Software Engineering.- 2016, 9, 2016.- p.296-317, http://www.scrip.org/journal/jsea, http://dx.do.org/10.4236/jsea.96021*
17. *Lavrischeva E. Generative and composition programming: aspects of developing software system families.- Cybernetics and Systems Analysis, Springer Volume 49, Issue 1 (2013), Page 110-123.*
18. *Lavrischeva E. M. Theory and practice of software factories.- Cybernetics and a System Analysis.- No. 6, 2011.- S. 145-158.*
19. *Theory and practice of software factories K. M. Lavrischeva, 2011, Volume 47, Number 6, Pages 961-972.*
20. *Pohl K., Böckle G., van der Linden F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005. DOI: 10.1007/3-540-28901-1.*
21. *Berger T., She S., Lotufo R., Wąsowski A., Czarnecki K. A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering, 39(12):1611-1640, 2013. DOI: 10.1109/TSE.2013.34.*
22. *Ekaterina Lavrischeva, Andrey Stenyashin, Andrii Kolesnyk. Object-Component Development of Application and Systems. Theory and Practice /Journal of Software Engineering and Applications, 2014, http://www.scirp.org/journal/jsea.*
23. *Lavrischeva E. M. Theory of object-component modeling software systems.- Preprint the ISP, 2016, 48 p. www.ispras.ru/preprints/docs/prep_29_2016_pdf.*
24. *Lavrischeva E.M. Ontology of Domains. Ontological Description Sofware Engineering Domain— The Standard Life Cycle, Journal of Software Engineering and Applications, July 24, 2015.*
25. *Lavrischeva Ekaterina. Ontological Approach to the Formal Specification of the Standard Life Cycle, "Science and Information Conference-2015", Jule 28-30, London, UK,www.conference.thesai.org.- p.965-972.*
26. *E. M. Lavrischeva. 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-may 21 2015, 101-119. (in Russian).*
27. *Lavrischeva E.M., Grischenko V.N. Assembling programming. Fundamental industry of program systems. – Nauk. dumka, 2009.–p.371 (in Russia).*
28. *Andon F..I., Koval G.I., Rjrotun T.M. and etc. Foundation engineering of quality PS.- K.: Akademperiodica.- 2007.-680p. (in Russian).*

APPENDICES

Figure 4: (2014)

Figure 5: (2016)