Journal of Software Engineering and Applications, 2016, 9, 296-317
Published Online June 2016 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2016.96021

Assembling Paradigms of Programming in Software Engeneering

Eraterina M. Lavrischeva

Moscow Institute of Physics and Technology (State University) MIPT, Dolgoprudny, Russia
Email: lavryscheva@gmail.com

Received 20 April 2016; accepted 24 June 2016; published 27 June 2016

Abstract

Assembling paradigms programming are based on the reuses in any programming language (PL) with the passport data of their settings in WSDL. The method of assembling is formal and secures co-operation of the different reuses (module, object, component, service and so on) being developed. A formal means of these paradigms creation with help of interfaces is presented. Interface IDL (Stub, Skeleton) is containing data and operations for transmission data to other standard elements linked and describes in the standard language IDL. Assembling will be realized by integration of reuses elements in these paradigms on the instrumental-technological complex (ITC).

Keywords

Paradigm, Assembling, Theory, Interface, Reuses, Object, Component, Service, Program Systems, Interoperability, Multilanguages, Reengineering

1. Introduction

The term *paradigm* was defined by R. Floyd (1978) and it is a theory and method for setting the style of the writing program. We consider the paradigm: the modular, object, component, service, aspect, etc. The essence of each paradigm-independent formal description of a software element can be used as reuses in other systems. The term *build* (assemble) was used by V. M. Glushkov (1974) in relation to programmes, whose role is similar to the conveyor assembly into the vehicle Ford. This term began to be used by many programmers to build large programs on the IBM framework and almost all come to a common definition of *Assembly programming* (1982). This programming is by a method of combining (integration, composition, aggregation) of the assemblies independent software elements with using the interface (Stub, Skeleton) to more complex software structure. Offered paradigms of programming are oriented on development of the complex program systems from the different formal program elements of these paradigms with the use of interface objects. A formal vehicle includes theoretical and applied aspects of planning the proper elements and operation of their integration in the complex program systems (PS).

Assembling polyglot elements is based on theory of fundamental type of data (FDT), arising up still in the 1970 years of past century in the works Dijkstra, Hoare, Wirt, Agafonov and so on and ales and standard of general types of data of ISO/IEC 11404-2007 (General Data Types—GDT) for the generation GDT ↔ FDT. A given theory is originally realized on the assembly polyglot in the APROP (1982) system. The facilities of support of specification of elements in the multiple programming languages PL, description of interfaces in the IDL language, saving them in library of prepared elements belong to them. Assembling elements of paradigms in the new PS is carried out by interfaces and the special operations of instrumental-technological complex (ITC). These paradigms are taught on the course education "Software engineering" for students and also some themes SE for performing by bachelors, by master's degrees, graduate students of faculty of cybernetics of the Kiev national university of Tara's Shevchenko and faculty of informatics of the MIPT. A bunch of theoretical and applied aspects is implemented in the ITC (http://7dragons.ru/ru) and the KNU program factory (http://programsfactory.univ.kiev.ua) as CASE-instruments Students can study, create, certify and accumulate or configure objects and components in ITC repository, as well as assemble them in PS [1]-[5].

Next, the paper considers the modular assembly, an assembly object, assembly components and Assembly service programming and general means and methods for assembly programming.

2. Paradigm of the Modular Programming

In 1970s years the modular programming, a basis of which is made by methods of decomposition of subject domain and integration modules in the complex structures appeared. The modules will realize some functions and will use the functions of other modules.

Module it is logically finished part of the program, executing a definite function. He possesses by such properties: completeness, independence, separate compilation, repeated use and so on. The modules accumulated in libraries of the modules, as prepared "details", from which the more PS going and are adapted to the new terms of environment of their treatment.

Such structures PS were designed top-down, the prepared modules got out from libraries of the modules and going to the new structures PS. In institute of cybernetics AN of Ukraine development of different approaches to automated was conducted assembling the programs, including system of the computer-aided of the programs manufacturing—APROP (1976-1987) [1] [2]. In her main constituents were: module passport with the interface (passed data and Call operators) specification, method of assembling the different PL modules and function of type of data conversion for class of widely used PL machines of ES. Base conception of the APROP system-interface as a PL communication and modules, on record in different PL (FORTRAN, PL/1, Algol-60, Assembler, COBOL). Main its objects: module, module-mediator, interface model of MIL (Module Interface Language), library of the modules and method of assembling the different PL modules [6]-[10].

2.1. Interface Communication of the Modules in PL

Every pairs of modules in different PL associates by interface as a module-mediator (stab in Corba). Communication of pairs of the functional polyglot modules was executed in the APROP system on OS of IBM-360. Such product differed from the traditional monolithic product by the expressly modular construction. In function of interface mediator entered: data communication through the formal and actual parameters; verification of accordance of their types of data, quantity and order of their location of parameters. If types of given parameters—unreeving (for example, integer or real), direct and reverse their transformation entered in the mediator function. The generation module-mediator contains the appeals to the elements of library of interface functions, which are executed on transition m one module to other and back.

Great number of three from the caused and causing modules in different PL and module-mediator united in the APROP system in aggregate-monolithic product on ES OS-360, intended for the decision of class of applied tasks. The reflection of formal and actual parameters entered in the mediator function, verification of accordance of passed parameters (quantity and location order), and also their types of data. A model chart of communication of modules in different PL is shown on the **Figure 1**.

The program is led on chart with, in which two calls are contained—Call ( ) and Call B ( ) with parameters. These calls "pass through" interface modules-mediators A' and B', which carry out functions of data conversion

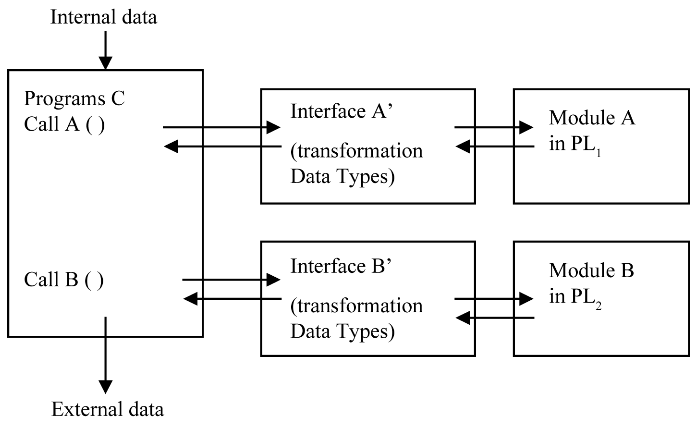


Figure 1. Chart of calls of the modules and in through the A' and B' interfaces.

and their transmission to the modules and B/After implementation of the modules and in results will be transformed back to type of the program C. If types of given parameters—not relevant (for example, is passed whole, and result—material or vice versa), direct and reverse their transformation enters in the mediator function. The generative module-mediator contains the appeals to the interface library elements, which are executed in the moment of transition of one module to other and back.

2.2. APROP System Functions

Executes the APROP systems next functions of union of the modules [1] [2]:

- treatment of the passport given modules in PL;
- analysis of parameter of the modules specification and drafting a task on treatment of irrelevant types of data, verification of rightness of the parameter passing on their quantity and on types of data in the PL class;
- type of data conversion in by the PL (b-Boolean, c-character, i-integer, r-real, a-array, z-record, etc.) generator of types of data with the use of interface library functions;
- Generation of the modules-mediators and drafting table of communication of pairs of components;
- Integration of pairs of the modules and their placing in structure of program aggregate;
- Translation and compiling the modules in PL as a prepared program structure;
- tracing interfaces and debugging functions of the modules in every pair of aggregate structure;
- testing a program aggregate on the whole;
- forming the programs of start of program aggregate and document.

In the complement of the APROP system the next components enter:

- Systems of programming (PL/1, FORTRAN, Assembler, language-constructor graphs);
- Treatment elements (modules in PL, Bank of the modules, Library of functions of data type conversion, language interpreter, intermodal interface);
- Automation facilities (treatment of the modules in PL, generation of the modules of communications, assembling the interlingua modules, testing the modules, testing interfaces, testing a complex aggregate from module);
- Forming an output result.

In this system it is realized assembling, based on two types of interfaces (intermodal and interlinguas) for every pair of the modules in PL. An idea of realization of intermodal interface in the APROP system is protected in the candidate's dissertation Grichenko V.N. (1991), method of assembling in the doctoral dissertation of author and in monographs [1]-[3].

Thus, *interface of the modules* as a communication of different types of objects in PL mean—the first home paradigm of interface in programming is realized in the APROP system.

Two types of interface are realized in the system—interface of the modules and the PL pair, interface between modules and interlingual interfaces.

Interface between modules of PS—it component for the generation of the interface modules-copulas of the interactive between itself modules.

Interlingual system *interface*—it is a component, containing a set of functions and macros in class of types of data and structures of the great number PL on their relevant transformation, described higher.

Interface of the modules, as a communication of different linguistic objects in PL mean—the first realization of interface paradigm in programming (1975-1982) [1]-[5].

The generation of the interface modules-mediators for the two objects with operators of converting passed data and control transfers enters in task of *interface* between different modules (there and back).

Assembling the prepared modules in the APROP system is based on aggregate of the modules in Bank of the modules, their passports and operators of assembling parts of the programs and complex systems [1]:

1) Operator of the Link assembling, questioner assembling two objects or count modules;

2) Link sag A ($A2$, $A3$ * $A4$)—to link the A, $A3$ and $A4$ modules in the segment structure and, where $A4$ is caused dynamically;

3) Link Prig B (($B1$, $B2$), $C = X$ ($C1$), $D = (Y, D1 = Y1)$)—to unite the $B1$ modules, $B2$, to them to add with and D with the $C1$ parameters. Y, $D1$.

4) Operator//The EXEC *module* $A1$//PL *Trans* $A1$…

5) To generate an interface mediator mod-interface generation for $A1 \cap A2$ and so on…

The rules of assembling determine compatibility of united objects, which contain description of functions for the concordance of different descriptions, are presented in their passports.

The process of assembling modules can be conducted by the hand, automated and automatic methods. As a rule, the last shift is impossible, is related to the not enough formal determination of the program reuses RC and their interfaces. A hand method is inadvisable, since assembling ready components RC is a large volume of actions, which carry rather conservative, by what creative nature. The most acceptable method—it is the automated assembling, when on the set specifications of the programs assembling by the standard rules of assembling heterogeneous objects is carried out.

The facilities which support a given method of assemble name instrumental by facilities of the assembling programming. The facilities of completion belong to them (unions of components in the more complex object); interface facilities of description and use of models of programming (aggregate of models of integration of different program elements).

Necessary terms of application of this method of programming are:

1) Presence of generous amount of various RC, as objects of assembling;

2) Passport system of objects of assembling;

3) Presence enough's complete set of standard rules of communication of objects, algorithms of their realization and facilities of automation of process of assembling;

4) Technologies by line with sequence of operations of the gradual making and establishment of communications between RC in case of formation of the system or PS.

The last condition means that the definite forms of PS presentation must exist as knowledge's about the subject domains, universal from point of planning and PS development. The main thing task of assembling – exposure of communication types, description of them in interface and realization as a mediator of interfaces between some modules and/or components, which secure their "docking" or communication in process implementation in some environment.

Assembly method. This method with interface was developed abroad in the projects MIL, SAA, IBM, Sun, Оберon, Corba and so on. The idea of assembling presently became model in class of traditional and modern PL. She is realized like in the adopted systems and is based on theory of unreeving type of data conversion in PL and is described in guidance's on application. New approaches to assembling are published in a number of works, including at 1. Bay (Co-operation of the polyglot programs, 2005) and so on.

The Corba system for the objective elements realized a universal manager of mediator—broker of objective queries, which links prepared objects-methods and components in any PL through mediators—stub (there) and skeleton (back). An interface mediator of objects is described in the new language IDL (Interface Definition Language). In him parameters of data communication are marked *in* and *out* between the heterogeneous objects in any PL. Data of prepared objects and RC contain the *type of data* declaration in corresponding PL and at their transmissions from one object to other they are checked on accordance of description in mediator of stub, them relevant parameters from skeleton. Thus, the result module paradigm is module on which you can run method of assembly using the interface.

3. Paradigm of Object Programming

At the turn of 1980s Grady Booch suggested object-oriented approach that changed the way of software development process. At that point of time structured programming approach had reached the *crisis of complexity*. As such, appearance of object-oriented programming (OOP) approach was seen as a way out of the crisis of the application complexity. Many analysts and programmers were looking to this approach with skepticism by seeing it as fashion and not as competitive advantage. The main thing that OOP gives is reducing the complexity of software artifacts, the ability to add and remove objects without any difficulty, hereby solving some aspects of the crisis of complexity [4]-[10].

The object-component theory was built on base of OOP formalisms, Frege triangle and Von Neumann-Bernays-Gödel set theory. This gives a chance to create common mechanism to generalize the notion of object with the appropriate properties and characteristics. The object is defined on the object level analysis involving logical and mathematical concepts of representation, function specification, Object Model (OM).

Every object comes to a set $O = (O_0, O_1, \cdots, O_n)$, where O_0 is the base object of domain. An object of domain O denotes a named part of the real world with a certain level of abstraction; it is described by Frege triangle (Denotation, Sign, Concept) (**Figure 2**).

On this figure, each object O is presented as $O_i = O_i$ (Na_i, Den_i, Con_i), where Na_i, Den_i, Con_i are a sign (name), denotation and the concept of object, respectively:

The sign sets a name of a certain essence in the real world;

The denotation designates an essence by sign;

The concept reflects semantics of the denotation, which is specified at levels of design of objects, bringing in the mathematical apparatus.

3.1. Levels of Logical and Mathematical Modeling of Domain

Design of domain model is performed at the four logical and mathematical levels of object definition (**Figure 3**):

1) *Generalized level* is used to determine the basic concepts of the given domain excluding their properties;

2) *Structural level* is used to determine the location of objects in the structure of the model and establish relationships between these objects;

3) *Characteristic level* serves for setting the general concepts and specific features of objects;

4) *Behavioral level* determines the behavior and modification of objects based on events that they create in their interaction with each other.

***Axiom* 2.1.** Domain area that is modeled from objects is an object itself.

***Axiom* 2.2.** Domain area that is modeled may be a part of another domain area.

When modeling an object from any given domain area, it has at least one property or characteristic, semantics and unique identification in a set of objects of that domain area and the set of properties (predicates) providing relationships between objects in the given domain area.

The **property** of the object is defined by a unary predicate that takes value of true for its external and internal features of the model.

Feature is a collection of properties (unary predicates), which are a subset of the set of selected system predicates which returns true values if the object implements such functionality.

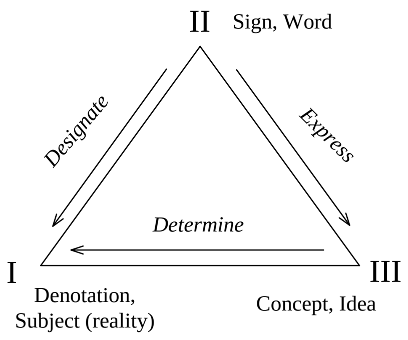


Figure 2. Frege triangle.

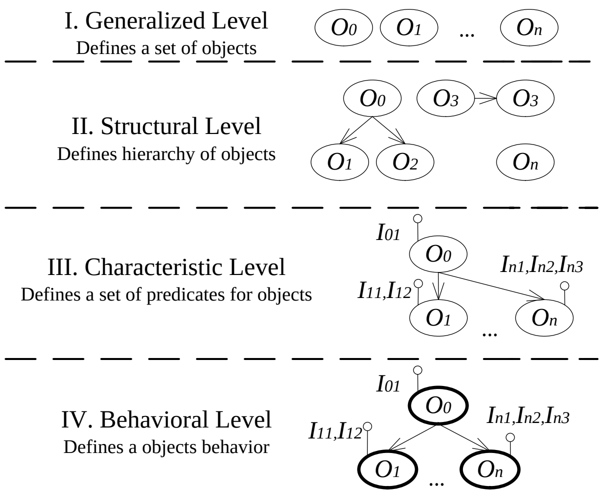

I. Generalized Level
Defines a set of objects
II. Structural Level
Defines hierarchy of objects
III. Characteristic Level
Defines a set of predicates for objects
IV. Behavioral Level
Defines a objects behavior
O0
O1
O2
O3
On
I01
I11,I12
In1,In2,In3

Figure 3. Example of object-interface graph. At each level of the design uses the logical-mathematical operations: ($\cup$, $\cap$, /, $\Diamond$, $\oplus$, –).

Relation is indicated by binary predicate over the set of objects and returns true on a given pair of objects. The main types of relationships are: the set—to the set, the element from the set—to an element of the set, an element of the set—to the set, the set—to an element of the set.

These relation types correspond to the following operations: generalization, specialization, aggregation, association, details of classification and instantiation. The type of relation 3 and 4 are IS-A and PART-OF. They are used for structural ordering of pairs of objects.

Designing of domain models. Set-theoretic concept lies at the foundation of model design. On the first design level, domain objects are detected. Then these objects are structurally arranged on the base of binary relation. On the generalized level an object is regarded as mathematical concept according to Von Neumann-Bernays-Gödel set theory.

On this level of design, the domain area as a set of basic functions of objects is formed associated with decomposition or compositional changes in variable objects and components. It consists of a number of functions that cover the transformation of denotations and concepts in the object analysis and the objects which include changes associated with an increase or decrease in the number of objects.

The result of *the generalized level* is a set of objects $O = (O_0, O_1, \cdots, On)$, where O_0 corresponds to the modeled domain area. The following is true for the O set:

$$\forall I\left[(I>0)\&(O_i \in O_0)\right] \tag{3.1}$$

On *the structural design level* each object is presented as a set or a particular element of the set. In this case, the expression (1) is transformed into the following form:

$$\forall i \exists j\left[(i>0)\&(j \ge 0)\&(i \ne j)\&(O_i \in O_j)\right] \tag{3.2}$$

Here, each object (except O_0) is the set of elements or certain elements of the sets, so set-theoretic algebra operations can be applied to them. This expression defines the PART-OF relation and instantiation. Thus, on this level classes, class instances and so on are defined. The properties of objects and relations are defined for these objects, together with classification, specification, etc.

On *the characteristic level*, each object corresponds to a concept. If $O' = (O_1, O_2, \cdots, O_n)$ is the set of objects of domain, and $P' = (P_1, P_2, \cdots, P_r)$ is a set of unary predicates related to the properties of objects in domain, then the concept of object O_i is the set of statements that are based on predicate P', assuming truth for the corresponding object. This means concept $Con_i = \{P_{ik}\}$ provided $P_k(O_i) = true$, where P_{ik} is a statement for the object O_i according to the predicate P_k. According to these rules, the properties and characteristics of objects are determined within a given abstraction of IS-A for object concepts.

Expression $A = (O', P')$ defines the algebraic system of objects concepts O' and predicates P', intended for objects analysis and identifying domain features.

***Axiom* 2.3.** Every object of domain has at least one property or characteristic, which defines the semantics and the unique identification of a plurality of domain objects.

Predicates P' contains following operations: 0-ary operations for constants, unary operations for the objects properties, binary operations to provide relationships between pairs of objects.

The behavioral level defines the sequence of object states and their processes of transition or reflection from one state to another. The relations between objects are formed on the basis of binary predicates that are associated with the objects properties, and the level of relation detailing between the classes of objects.

The main objective of every level is to describe the internal and external characteristics of objects, which it is necessary for organization of connection between them.

The concept of a class is replaced by the notion of set. If the object is part of another object, it is determined by the set. However, not every object is an element of any other object (class). For example, an object that is responsible for the whole of a particular OM is not a part of any other object in this OM. The definition of the object is formulated by: every object is with necessity a set or an element of a set.

The regular operations are executed over the set of objects: association, intersection, difference, combining, symmetric difference, Cartesian product. Specification of object in a given concept are class (a set of objects); an instance of a class (an object that is an element of a certain set), consolidated class (a set which is a direct sum of several other sets); crossing-class (a set that is a common part of other sets), aggregated class (a set which is a subset of the Cartesian product of several other sets).

Establishment of a particular domain model has an iterative nature and begins with the definition of domain as the start object. At each iteration, analytic functions that approximate the structure and properties of domain objects are applied until the final model is built.

3.2. Object Analysis of Domains

The modeling of objects is performed in the following system:

$$\Sigma = \left(O', I', A', P'\right), \tag{3.3}$$

where $O' = (O_1, O_2, \cdots, O_n)$ is a set of objects, I is a set of interfaces of O'; $A' = (A_1, A_2, \cdots, A_n)$ is a set of operations over elements of the set O; $P = (P_1, P_2, \cdots, P_r)$ is a set of predicates to define the properties of the object concepts (for example, $Con_i = (P_{i1}) = true/false$). Each of the operations A' has a certain priority, arity and associates with the appropriate valid descriptions (Na_i, Den_i, Con_i) of object signs, denotations and concepts and the set of operations $A' = \{decds, decdn, comds, comdn, conexp, Conner\}$. In other words, *decds*, *decdn* are decomposition; *comds*, *comdn* are composition and *conexp*, *Conner* are contraction.

Theorem 2.1. The set of operations A' of object algebra is a system of operations on the four-tiered representation of the model of the object domain.

The result of the structural ordering of object model OM is graph $G = \{O, I, R\}$, defined on the set of objects O, interfaces I and relations R between objects. Conditions of constructing the graph G:

The set of vertices O displays all domain objects one-to-one;

For each vertex, there must be at least one interface $I_k \in I$ and relation, owned by a set of relations R;

There is at least one vertex that has the status of a "*set of objects*" and reflects the domain area in general with accordance to the axioms 2.1, 2.2.

Built graph G is complemented by interface objects and structurally ordered to control the completeness and redundancy of graph elements and eliminate duplicate elements (**Figure 4**).

The set of objects-functions O is associated with a set of implementation methods on remote domain objects. During specification, the objects are linked through interface objects from the set I. It means vertices of the graph G belong to one of the two types—functional or local O and interfaces I.

In the graph, interface objects correspond to functions describing data exchanged between objects (**Figure 5**), methods of data transmission through RPC operators, RMI operations and conversion of some data not relevant in terms of their order of execution platforms and formats.

Graph G set of domain objects and interfaces form the OM. The objects of OM are represented by the general and individual properties or the external and internal characteristics. The check of object properties is performed by applying instantiation operations. Interface objects can be input (In) and output (Out) from the set of interfaces $I = (In\ (O_k), Out\ (O_k))$, *i.e.* the input interfaces $In\ (O_k)$ and output interfaces $Out\ (O_k)$.

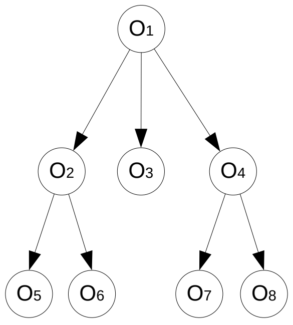


Figure 4. The object graph *G*.

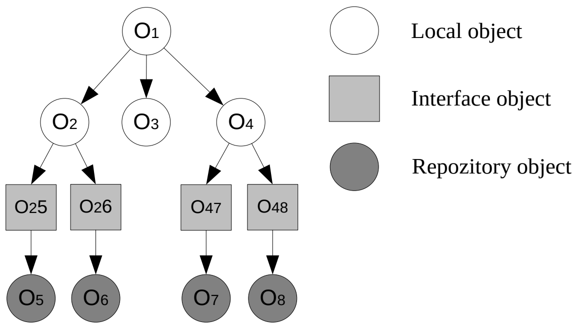


Figure 5. The object-interface graph *GI*.

To define the semantics of the object graph *G*, IDL interface language is used (proposed by OMG). OMG was first implemented in CORBA system, coupled with methods for constructing and developing shared objects [9].

To the aggregate of the set-theoretic operations include: Ω = (∪, ∩, /, ◊, ⊕, –). Algebraic system for structurally-ordered level is: Σ = (*S*, Ω). Objects defined at the synthesis level are provided as sets of elements or a specific set of algebraic systems Σ = (*O*′, Ω) structural level.

Input and output parameters, which are exchanged between functional objects, are described by myltilanguages (*C*, *C*++, *C*#, Pascal, Ruby and etc.). On **Figure 4**, objects O_2 and O_5 yield an interface object $O_{2.5}$, which secures data exchange for conducting calculations. If types of passed parameters in the interface object are irrelevant (e.g., input parameter is integer and output is real), generative functions are created for the transformation integer ↔ real [10]-[15].

In graph *GI* (**Figure 5**) you can build a program P_1-P_6 using operation ∪, function interface *In* and the operator *link*:

1) $P_1 = O_2 \cup O_5$, $link\ P_1 = In\, O'_{25}(O_2 \cup O_5)$;

2) $P_2 = O_2 \cup O_6$, $link\ P_2 = In\, O'_{28}(O_2 \cup O_8)$;

3) P_3;

4) $P_4 = O_4 \cup O_7$, $link\ P_4 = In\, O'_{47}(O_4 \cup O_7)$;

5) $P_5 = O_4 \cup O_8$, $link\ P_4 = In\, O'_{48}(O_4 \cup O_8)$;

6) $P_0 = (P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5)$. (3.4)

Results from the linkage of two objects of a graph (e.g., $link\ P_2(O'_{25})$ is an interface object $In\, O'_{25}$. There are many input interfaces coincides with the set of interfaces of the target object, and a plurality of source interfaces with multiple *input* interfaces of the object transmitter.

Axiom 1.2. Advanced graph G with interface objects that are structurally ordered (top), controlled completeness, redundancy and backup elements.

Objects can have multiple interfaces that can inherit the interfaces of other objects in the case when they provide the data to the multiplicity of interfaces.

Many of the objects and interfaces of the graph relates to common characteristics of objects in OM. They are considered to be reliable if the following condition holds: each internal characteristic is equivalent to the external characteristics of the object. If this condition is not met, the element is removed from the set and from the graph respectively.

Formal association operations of objects and interfaces are as follows:

$O_k \in O, In(O_k)$ are a set of input (*In*) interface objects;

$O_k \in O, Out(O_k)$ are a set of output (*Out*) interface objects.

Classes of objects. When objects are combined according to the general characteristics of classes in the OM, it has following form: *OM* = (*Oclass*, *GC*), where *Oclass* = {*Oclass*^{i}} is a set of object classes for functions or methods with common properties; *GC* is an object graph that reflects connections and relationships between classes and instances.

Each class is represented as

$$Oclass^i = \left(ClassName^i, Meth^i, Field^i\right\}, \tag{3.5}$$

where *ClassName*^{i} is the name of the class; $Meth^i = \left\{Meth_j^i\right\}$ is a set of its methods; $Field^i = \left\{Field_n^i\right\}$ is a set of properties that determine the state of the class instances.

Every $Pfield_n^i \in Pfield^i$ has methods $get\left\langle Pfield_n^i\right\rangle$ and $set\left\langle Pfield_n^i\right\rangle$ for writing and reading values of the corresponding properties as attributes and interfaces of OM and component models in which the objects methods are represented by software components.

The set of methods provided as:

$$IMeth^i = IMeth \cup \left\{get\left\langle Pfield_n^i\right\rangle\right\} \cup \left\{set\left\langle Pfield_n^i\right\rangle\right\}, \tag{3.6}$$

which corresponds to the interface *Ifunc*^{i} consisting from the methods from *IMeth*^{i}.

3.3. Description of Interface Objects

Formalism of description of the IDL interfaces is presented in OMG CORBA. In it, the interface mediators (stub, skeleton) contain the passed data definition between PL-objects as stub for a client and skeleton for the server and operations of data communication between objects [10].

The description of operations of data communication includes:

Name of interface operation;

List of parameters (zero or more);

Types of arguments and results, otherwise—void;

Handling parameter for description of exceptional situation, etc.

IDL language is used for interface specifications (such as stub or skeleton). Interface options have the following description: **in**—input parameter, **out**—output parameter, **inout**—compatible option.

The interface specification for one object can be inherited by another object and this description then becomes the base. The interface inheritance mechanism consists of saving names of objects without resizing them. It concerns description of operations, which must have unique denotations.

The names of operations can be used dynamically during implementation of *skeleton* interface.

A type is described in the TD class, which is passed through parameters of the RPC operators, RMI, as well as with WCF, .NET protocols, etc. TD is described in one of OOP languages (C#, VBasic, Pascal, and so on).

Input and output interfaces for the P_1 and P_2 programs (**Figure 6**) have different semantics, but identical syntactic description in a certain PL. Data communication between these programs and P_3 is carried out through the functions F_1 (.), F_2 (.) and the **In** and **Out** interfaces, which perform TD transformation of data passed between P_1, P_3, and P_2, P_3.

The described formalism of interface is presented not only in the CORBA system, but also in IBM OS, Microsoft and so on. It is based on libraries with data type conversion utilities, which are used during integration of heterogeneous program resources (objects, components, services). Thus, the result object paradigm is object on which you can run method of assembly using the interface.

4. Paradigm of the Component Programming

The component programming is based on Ready Components (RC) reuses. Creation of the new PS from RC is considered as a strategically way to the rise of productivity labors in programming and providing his quality. He

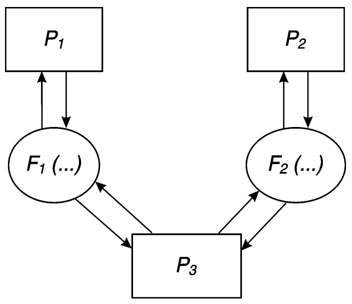


Figure 6. Chart of function calls between objects.

is development of the module and object to the components at direction of generalization of collective technology and principle of Reusability, as bases of modern industrial technologies and factories of the programs. Indicated types of program elements are united by the general principle of reusable, which is used in much spheres of activity of different categories of computer specialists. But, without regard to that ready RC and components a generous amount is accumulated, the component programming of the new systems remains problematic, because for a few years the generation of computer facilities changes and it requires reengineering considerable part of the prepared programs that causes the considerable duplication of functions. Such situation was folded as a result of absence of standardized approaches to construction of the programs from the prepared program elements, and also creation of elements, and related to absence of necessary theoretical and methodological base for decision of tasks of creation of the PS from the prepared elements [15]-[17].

It large attention is spared and abroad. She is explored in the state programs of Japan, Great Britain, the USA and so on, and also in the peace corporations (IBM, Microsoft, Intel, Hewlett-Packard and etc). Problem of reuses are considered at the international conferences during decades. In reuses are laid large facility for the receipt of income in case of development from them of complex PS, as financial products.

That is why development of theoretical, methodological and engineering bases of component programming for the decision of this actual tasks, and also contemporaneity tasks such, as creation on the component basis of the PS, Web-applications, modern information systems and technologies, is a very important and perspective problem.

4.1. Basic Concepts of Component Programming

Component is understood by us as a PL-independent self-relevant of software product that provides execution for a certain set of application functions and services of applied domain, which can be accessed with remote Calls [18].

RC as an element of typical repeated decision in PS is necessary typical architecture, descriptions and attributes, which are given in the interface part and are used at the exchange by data for co-operation of components in the different environments. That is given thus component, becomes an indivisible and encapsulation of object, which satisfies to the functional requirements, and also requirements in relation to architecture of the system and implementation environment component programs.

Basic operations of components are:

- specifying components and their interfaces (pre- and post-conditions, which must be satisfied by caller them) in such languages as IDL, API, WSDL etc.;
- maintenance of components and reuses into the component repository for their future integration
- collection RC into Applications, Domain, PS etc.

Component program or PS from components—it is an aggregate of components, that will realize the functional and not functional requirements to her, is built on rules of configurations of collective type with providing co-operation.

Model domain is built in objects which can be transformed to the components with the use of mathematical vehicle: models of component and interface, component environment, external and internal component algebra and algebra system of transformation of types of given components for their co-operation [12]. Component construction PS from ready RC from the different libraries gives the considerable cutback of spending and improves PS quality [17].

4.2. The Model Interface and Component

A model of component is investigation of generalization of typical decisions and has a kind [18]:

$$C = (CNa, CFa, Cln, CIm, CSe), \tag{4.1}$$

where *CNa*—unique identifier of component;

CFa—interface of providing functions of management by copies of component;

CIn = {Cln_1, ···, Cln_k}—set of interfaces, related to the component;

CIm = {*CImj*}—a set of realization of component;

CSe = {*CSer*}—a set of general system services.

The set of the interfaces components *CIn* = {*CInIn* ∪ *CInOut*) consists of entrance *CInIn* and the initial (output) *CinOut*—initial (output) interfaces (realization is in other components) as parameters of interface. That is the component has entrance interfaces at own realization, and initial interfaces of realization other component. Each of them has the proper model

$$CIn^i = (InNa^i, InFu^i, InSp^i) \tag{4.2}$$

where $InNa^i$—name of interface, $InFu^i$—functionality (aggregate of methods), $InSp^i$—interface specification: type, constants elements of signature of methods, descriptions and others like that declaration.

The *CFa* function interface determines the methods and appeal to the specimen of components (search, choice, elimination and others like that).

Every realization *CImj* ∈ *CIm* is a set by model:

$$CIm^j = (ImNa^j, ImFu^j, ImSj), \tag{4.3}$$

where $ImNa^j$—identifier or name realization, $ImFu^j$—functionality, $ImSp^j$—description specification implementation condition, which dependence from the definite platforms).

A necessary requirement of existence of component is been by condition of his integrity:

$$\forall CIn^i \in CInI\ \exists CIm^j \in CIm\left[Pr\left(CIn^i\right) \subseteq CIm^j\right], \tag{4.4}$$

where *Pr* (CIn^i) means functionality from realization of CIn^i interface methods.

Axiom 2.2. For the union of two heterogeneous components *C*1 and *C*2 it is necessary *the* terms exist, if *CIni*1 ∈ *CInO*1 and *CInk*2 ∈ *CInI*2 such, that *S* (*CIni*1) = *S* (*CInk*2) & *Pr* (*CIni*1) ⊆ *CImj*2, where *S* (*sign*).) means signature of the proper interface.

Tne Model Component PS

This model P*S* is identical to the model of program component with that particulte that by its elements there can be the programs, as independent components.

$$PS = PSLm\{Lm^1, \cdots, Lm^n\}, R\{R^i, \cdots, R^m\}, PSLn\{Ln^1, \cdots, Ln^k\},$$

where $PSLm\{Lm^1, \cdots, Lm^n\}$—a set of the component realisation RC, PS; $R\{R^i, \cdots, R^m\}$—a set of predicates; $PSLn\{Ln^1, \cdots, Ln^k\}$—set of interfaces of components and programs.

The operations of the set *R* can answer the union or RC configuration in some complex structures of the programs and PS from the set of components interfaces.

Components or RC can change, to be replaced new functionally by similar, equivalent or identical RC with purpose of receipt of new variants of the programs product (PP) after axioms.

Primary objective PS this communication (assembling) of components, and also change some RC identical or equivalent in the PS structure. The communication has it арність and can consist of set of data, which enter to the class of interfaces of the marked class RC. This operation is executed after model of component (3.1) and interface (3.2), which contain the predicates, which determine a condition of data communication to other components.

Between components the relations can exist: inheritance enterharance, *ecsemplirations*, *contract*, *union* (*communication*). All relations are determined on interfaces and predicates after such maintenance.

Relation of ecsemplirationї. $CInski^j$ = ($IInski^j$, $InFu^i$, $ImFu^j$) describes the definite exsemplar component *C*, where: $IInski^j$—unique identifier of exsemplar, $InFu^i$—interface functionality Cln^l ∈ *CIn*, $ImFu^j$—element, that secures implementation *realization* of $CIm^i \in CIm$.

Relation of contract. Between the C_1 and C_2 components the relation of contract exists

$$Cont1_{2i}^{m} = \left(CInO_1^i, CInI_2^m, IMap_{12}^{im} \right),$$

where $CInO1^i \in CIn1$—initial interface of first component, $CInI2^m \in CIn2$—entrance interface of second, $IMap_{12}^{im}$—mapping of accordance between methods, which enter in the complement of both interfaces taking into account signature and passed types of data. If the C_2 component has realization of the $CInI_{2m}$ interface, implementation of functionality of the $InFu1^i$ interface $CInO1^i$ *is* secured.

Relation of interconnection. If between the C_1 and C_2 components the relation of the $Cont1_{2i}^{m}$ contract exists, between their copies $CIns1kij = (IIns1ki^j, InFu^i, ImFu^j)$ but $CIns2p^{mq} = (IIns2p^{mq}, InFu^m, ImFu^q)$ the relation of communication in regard to the $Cont1_{2i}^{m}$—contract, which is described in kind exists

$$Bind\left(IIns1ki^j, IIns2p^{mq}, Cont12^{im} \right).$$

Relation of interoperability. If the PS components located in the distributed environment and they enter to the set of components, RCs to repository PS, a model of co-operation, definite on the set of interfaces of ready RC and remote Calls is used [18].

4.3. Model of Component Environment

***Definition* 3.1.** Component environment—it is a definite set of components, which co-operate between it through interfaces and services, which regulate communication between components and interfaces, which are saved in proper repositories.

The model of component environment has next expression:

$$CE = \left(CNa, InRep, ImRep, CSe, CSeIm \right), \quad (4.5)$$

where $CNa = \{CNa^i\}$—a set of names of components, which enter in the complement of environment; $InRep = \{InRep^i\}$—epository interfaces of environment components, $ImRep = \{ImRep^j\}$—repository *realization*, $CSe = \{CSe^i\}$—interface of system services, $CSeIm = \{CSeImr\}$—a set of realization for the system services.

Every element from *InRep* it is two (CIn^i, CNa^j), where CIn^i—interface of components, CNa^i—component name for which the interface exists. Every element with *ImRep is* like described (CIm^j, CNa^i), where CIm^j—realization, which is described by expression (3.4), and CNa^j—name of component, to which this realization belongs.

A component environment is considered as a set of servers of PS, where components-containers, the exsempers of which will realize functionality of component are opened out. Container intercommunication with server is secured through the standardized interfaces (*CFa*). Interconnection between different RC, which are unfolded in the different servers, are secured by realization of the *CSe* interface.

The condition of integrity of the component program consists in existence for each component *C*1 from the *CE* initial interface *CInO*1*i* and component *C*2 with the proper entrance interface *CInI*2*m* and contract *Cont*12*im* = (*CInO*1*i*, *CInI*2*m*, *IMap*12*im*), that enters in the complement of the set *Cont*.

***Defenition* 3.2.** Program system (PS) it is a set RC, functions (objects), interfaces and data. Will give the PS model with the use of components in such kind:

$$M_{PS} = \left\langle CSet, P, PC, M_f, M_s, M_{int}, M_d \right\rangle, \quad (4.6)$$

where $CSet = \{C_1, \cdots, C_n\}$—a set of components;

P—some predicate from the C_i belonging to PS;

PC—specific predicate, which determines, that C_i satisfies to the definite function from the great number M_f;

$M_f = \{F_1, F_2, \cdots, Fr\}$—a set of functions (methods) domens, also comes forward as a model of PS descriptions;

$Ms = \{Ms^{in}, Ms^{out}, Ms^{inout}\}$—a set of services from communication with entrance Ms^{in} by initial $Ms^{out}t$ and server data Ms^{inout} of PS;

M_{int}—model of interfaces of interconnection of objects between itself;

M_d—a set of data and metadata of the subject domain PS.

The sets M_f, M_s and M_i *are* related to the interface data (type of *in*, *out*, *inout*) and points of variants.The condition of integrity of the component program consists in existence for each component C_1 with *CE*, that the initial interface *CInO*1*i* has, component *C*2 with the proper entrance interface *CInI*2*m* and *Cont*12*im* contract = (*CInO*1*i*, *CInI*2*m*, *IMap*12*im*) enters in the complement of the set *Cont*.

For description of functionality of components in different PL with the proper types of data in them are used. In case of union, integration of such components the task of transformation of types of data passed by interface for the relevant converting of types of data according to model of integrated environment makes up one's mind.

Two approaches exist in case of decision of RC integration problem:

1) Application of *model of co-operation of* pair of polyglot components in PL in environment of the distributed systems by their composition through the module mediator (stub, skeleton) with the use of *interlinguas* and intermodal interface [11];

2) RC interface description in the IDL language after the *In* parameters, *Out*, *Inout* for the task of data at the data exchange between itself [9] [10].

4.4. Transfer Object Model to Component Model

Component program or component application is a set of components that implement the functional and non-functional requirements and is built according to the rules of component configurations within the component model framework.

According to the formal definition of the domain model, it is a source of transformative shift from objects to components using formal mathematical tools of modeling: models and component interfaces, component-based applications, the component environment, internal and external component algebra construction [9]. This apparatus is used for formal development of component applications using reusable assets from different libraries with significant reduction in costs and improvement of the quality of future products [2] [7].

Component model emerges from generalized solutions to the nature of objects. The methods of objects are converted to the component architecture, structure, properties, and characteristics.

There are two approaches to solving the problem of integration of reusable components:

1) Use of the interaction model through the intermediary module (stub, skeleton) using cross-language and intermediate interface [1] [2];

2) Description of the reusable component interface in the IDL language with **in**, **out**, **inout** parameters to specify the data values.

In the theoretical aspect, multilingual component integration is based on formal mappings (presented as a superposition of base mappings).

4.5. Modification Applications from Components

The system is assembled from reusable components by modifying or adding their interfaces and/or implementations [2]-[5].

The general component algebra includes external, internal and evolutional algebras:

$$\Sum=\{\varphi_1,\varphi_2,\varphi_3\}=\{CSet,CESet,\Omega_1\}\cup\{CSet,CESet,\Omega_2\}\cup\{CSet,CESet,\Omega_3\}, \qquad (4.7)$$

where φ_1 is the external component algebra, φ_2 is the internal component algebra, φ_3 is the evolution component algebra.

External component algebra $\varphi_1 = \{CSet, CESet, \Omega_1\}$, where

CSet is a set of components *C*, *CESet* is the environment *E* with set of components *C* and interfaces *Int*.

$\Omega_1 = \{CE_1, CE_2, CE_3, CE_4\}$ represents algebra operations:

CE_1—operations of component processing;

CE_2—initialization operations;

$CE_3 = CE_1 \cup CE_2$—assembling operations;

$CE_4 = CE_1 \backslash C$—operation for extracting component *C* from its environment;

$C_2 - CE_2 = C_2 \oplus (CE_1 \backslash C_1)$—substitution operation.

Internal component algebra: $\varphi_2 = \{CSet, CESet, \Omega_2\}$,

where *CSet* = {*OldComp*, *NewComp*} is a set of old components *OldComp* and a set of the new components *NewComp*.

The set *OldComp* = (*OldCName*, *OldInt*, *CFact*, *OldImp*, *CServ*) includes interfaces, implementations in the server environment;

The set *NewComp* = (*NewCName*, *NewInt*, *CFact*, *NewImp*, *CServ*) includes interfaces, implementations for these components;

Ω_2 = {*addImp*, *addInt*, *replInt*, *replImp*}, where *addImp* denotes the operation of adding an implementation; *addInt* is the operation for adding an interface; *replImp* is the operation of the substitution of a component implementation, *replInt* is the substitution of a component interface.

Component Evolution Algebra. Reuse is the basis of the components evolution. Methods of transformation are divided into two types: methods that change the functionality and behavior of the components and methods that are associated with non-functional changes. The first type includes changes in the interface (changes in interface signatures, adding a new interface) and the implementation (changing algorithms and logic, replacing and adding realizations) and others. The second type includes changes related to non-functional characteristics of the application (reliability, efficiency and mobility), languages and execution platforms.

Component evolution algebra is φ_3 = {*CSet*, *CESet*, Ω_3}, where $\Omega_3 = \{O_{refac}, O_{Reing}, O_{Rever}\}$ is a set of component evolution operations; O_{refac} is the refactoring operation, O_{Reing} is the reengineering operation, O_{Rever} is the reverse engineering operation for a certain component.

Component refactoring model is as follows:

$$M_{Refac} = \left\{O_{Refac}, \left\{CSet = \left\{NewComp^n\right\}\right\}\right\}, \tag{4.8}$$

where O_{Refac} = {*AddImp*, *AddNImp*, *ReplImp*, *AddInt*} is the refactoring operation, the pair (*CSet*, O_{Refac}) is an element of the evolution component algebra.

Components of evolution algebras are built on base of the semantics terms and requirements on features refactoring implementation

$$O_{Refac} = \left(CSet, Ref\right),$$

where *CSet* = $\{C_n\}$ is a set of components, and *Refac* = {*AddImp*, *AddInt*, *RelImp*, *AddInt*} is a set of refactoring operations. *AddImp* adds a new implementation of the existing interface; *NewComp* = *AddImp* (*OldC*, *NewCIm*, *NewCInO*) is the operation of adding a new component, $NewCInt = OldCIn \cup NewCInO^s$ is the operation of adding output interfaces for a new implementation, $NewCImp = OldCIm \cup \{NewCImp\}$ is the operation of adding a new implementation.

Theorem 3.2. Algebra component refactoring $\sum^{refac}$ = (*CSet*, *Refac*) is complete and consistent.

Model for component reengineering is as follows:

$$M_{Reing} = \left\{O_{Reing}, \left\{CSet = \left\{NewComp^n\right\}\right\}\right\}, \tag{4.9}$$

where O_{Reing}= {*rewrite*, *restruc*, *adop*, *conver*} are reengineering operations, the pair (*CSet*, O_{Reing}) is an element of evolution component algebra.

Reengineering algebra components $\sum^{Reing}$ = (*CSet*, *Reing*) are used in case of component integrity violation or changes in their functionality.

Axiom 3.5. For $\exists OldInt \in OldInI$ there exists an adding operation of input interfaces and corresponding functionality.

This operation is associative and commutative and observes integrity of the component.

Operation *AddImp* denotes adding a new implementation of the input interface, which is not included in the set of component interfaces:

$$NewC = AdNIm\left(OldC, NewImp, NewCInt\right).$$

These operations are associative and commutative. The integrity of the component is retained.

Operation *ReplIm* is the replacement of an existing implementation by a new one without changing the input interface: *NewC* = *ReplIm* (*OldC*, *NewCImp*, *NewCInt*, *OldCImp*, *OldCInt*).

Lemma 3.1. Operation of replacing the existing implementation with a new one given terms and semantics, mentioned above, preserves the integrity of the component.

Operation *AddInt* is adding a new input interface for an existing implementation.

Lemma 3.2. Operation of adding a new input interface for an existing implementation given terms and semantics, mentioned above, preserves the integrity of the component.

Reverse engineering model: $M_{Rever} = \left\{ O_{Rever}, \left\{ CSet = \left\{ NewComp^{n} \right\} \right\} \right\}$, (4.10)
where O_{Rever} = {*restruc*, *conver*}, the pair (*CSet*, O_{Rever}) is an element of evolution component algebra. $\sum^{Rever}$ = (*CSet*, *Rever*).

In reverse engineering, reengineering operations can be used: *Reeng* ⊂ *Reverse*. The feature of the set *Reverse* is that its operations are not completely defined, but rather partially. It is because the reverse engineering means complete alteration of the program system using components.

Overall, the component algebra $\sum^{Reface}$, $\sum^{Reign}$, $\sum^{River}$, as well as models R_{Refac}, M_{Reing} and M_{Rever} constitute the formal apparatus of evolution algebra from models, operations and methods for the evolution of the components. Thus, the result component paradigm is the component on which you can run method of Assembly using the interface.

5. A Common Assembling Model and Methods for All Paradigms

Multilingual RC assembling model. Let *CSet* = {C_i} a set of components (or object, module) written in multiple programming languages. During their interaction, components C_i are exchanging data. Each pair of components C_i and C_j may be equivalent provided they have the same semantic structure and type, or non-equivalent otherwise. In the latter case their conversion is required using functions represented by mappings:

$$FN_{ij} : N_i -> N_j, FT_{ij} : T_i -> T_j, FV_{ij} : V_i -> V_j \tag{5.1}$$

with FN_{ij} establishing correspondence between the names of variables, FT_{ij} describing the equivalent mapping of data types, FV_{ij} implementing the necessary conversion of data values.

Problem of variables replacement FN_{ij} is solved by ordering variable names (for example, in the configuration description for the components in question). Mapping between data types FT_{ij} is based on the transformation of data types, each of which is represented by an abstract algebra and algebraic system $T = (X, \Omega)$, where X is a set of values that can make a change of this nature, and Ω is a set of operations over these variables. Reflection FV_{ij} is applied in case types T_i and T_j are not equivalent (e.g., the conversion of an integer value to real) [17] [18].

Conversion from the type $T_i = (X_i, \Omega_i)$ to the type $T_j = (X_j, \Omega_j)$ is meant as a conversion, in which the semantic content of operations from Ω_i is equivalent to content of operations from Ω_j. These conversions are available for common data types in most programming languages as described by the ISO/IEC General Data Types (GDT) standard 11,404 - 2007; it provides mechanisms for generating FDT from GDT [2].

The problem of component interconnection occurs when assembling them into compound structures. To solve this problem, the set of reflections is built for different types of method calls, in order to establish a correspondence between the set of actual parameters $V = \{v_1, v_2, \cdots, v_k\}$ of the object and set of formal parameters $F = \{f_1, f_2, \cdots, f_l\}$ for components. The problem of constructing data types using algebraic systems for basic data types used in various PL and isomorphic mapping between algebraic systems are considered following [2] [3].

Implementation of interfaces. Interface is based on transformation of irrelevant type of objects in different PLs, passed through mechanisms of formal and actual parameters. The data, related to general data types and fundamental data types can correspond to the request parameters of other objects. They may be incompatible between themselves because of the differing platforms, number of parameters sent, and varying compiler implementations of data types; therefore, they require the proper transformation [15] [16].

Different programming languages do not have a common implementation of fundamental data types and general data types, which are described in ISO/IEC 11,404 standard.

Fundamental types are:

Simple data types (real, integer, char, etc.);

Structural data types (array, record, vector, etc.);

Complex data types (set, table, sequence, etc.).

They are used by all programming languages and are implemented by translators.

General data types are:

Primitive data types (character, integer, real, complex number, etc.);

Aggregate data types (enumeration, pointer, set, bag, sequence, etc.);

Generated data types (which emerge as a product of data type generator from one or several data types);

Reusable components (reuse, artifact, object, component, service, etc.) are described in a programming language using one of the standard WSDL, Grid or IDL interfaces. They are saved in the RC repository and interface repository.

For data type transformation from one PL into other, three libraries are developed:

GDT library;

Library for reflection functions GDT ↔ FDT;

Library for functions for data transformation $PL_i \leftrightarrow PL_j$;

Intermediate libraries for functions and routines for transforming standard data types within a certain environment (CTS, CRL, FCL)

Theoretical integration of elements paradigm (module, object, component, service and so on) are based on the formal reflections, given as superposition of base reflections. Every component has passport data on the WSDL for interface (**Figure 7**).

In the given time problem of transformation of data it is offered in standard of general types of ISO/IEC 11404-2007 GDT data. By us GDT generation to FDT and transformation of them by *primitive functions of the* special libraries of the VS system machineries are offered.Net (CLR, CTS and CLS and others like that).

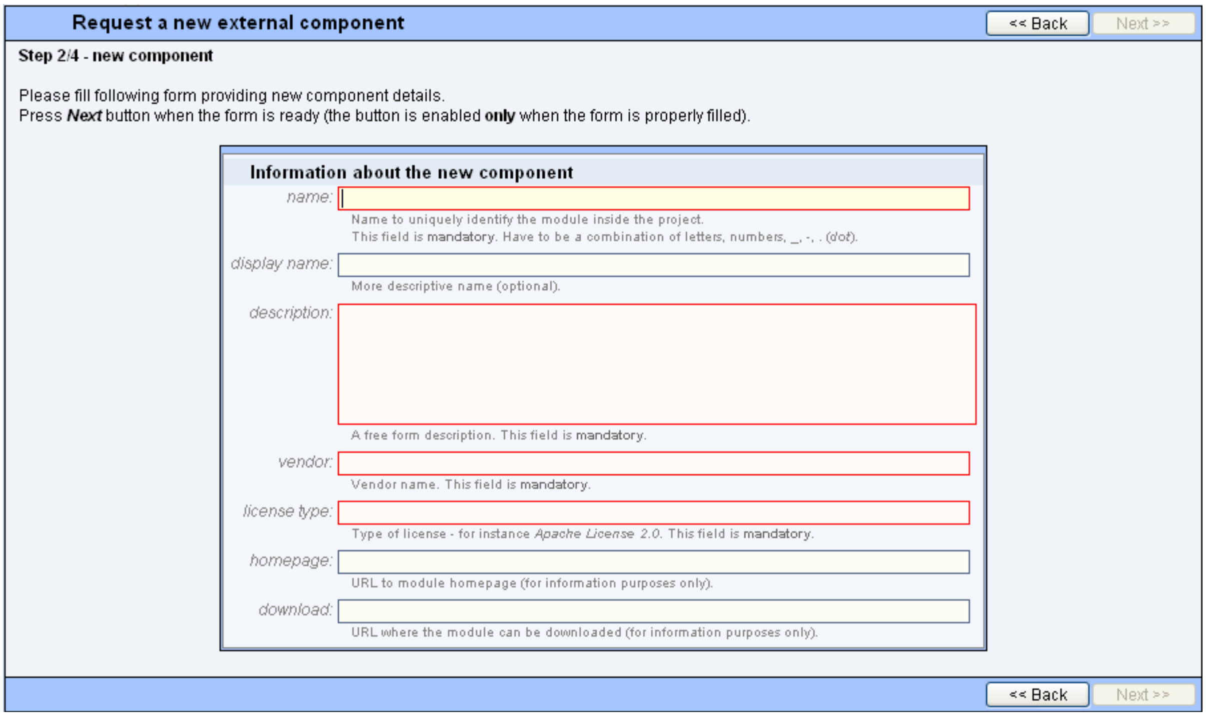
He is used for certification of prepared components and saving them in repository of student factory of the programs http://programsfactory.univ.kiev.ua)

From the formal specification data, by which are exchanged every pair of the C_i and C_j components, can be equivalent, if they have an identical semantic structure and type, otherwise—not equivalent.

5.1. Theory and Method of Assembling Elements Paradigms in the PL

The method of assembling includes the mathematical theory determinations of communications (from and on the management) between the different language modules and generation of the interface modules-mediators for every pair of the modules. Communication of the modules is described by operators of call of the CALL type with the great number of actual arguments (*in*, *out*, *input*), passed other, linked functionally to the module. These data is described in the module and interface [3] [5].

The main task of communication of pair of the different language modules consists of decision of task of their co-operation. The essence of this task consists of construction mutually of univocal correspondence between the great number of actual arguments of the $\{v^1, v^2, \cdots, e^{ve}\}$ defiant module and great number of the formal parameters $F = \{f_1, f_2, \cdots, f_{k1}\}$ caused module and reflection of their data by the algebraic systems.



Step 2/4 - new component

Please fill following form providing new component details.
Press ***Next*** button when the form is ready (the button is enabled **only** when the form is properly filled).

Information about the new component

name:
Name to uniquely identify the module inside the project.
This field is mandatory. Have to be a combination of letters, numbers, _, -, . (*dot*).

display name:
More descriptive name (optional).

description:
A free form description. This field is mandatory.

vendor:
Vendor name. This field is mandatory.

license type:
Type of license - for instance *Apache License 2.0*. This field is mandatory.

homepage:
URL to module homepage (for information purposes only).

download:
URL where the module can be downloaded (for information purposes only).

Figure 7. Standard of component data definition in WSDL.

The algebraic systems are built in class of primary types of data of the PL ($t = b, c, i, r$) complex types of data ($t = a, z, u, e$) and possible types of their transformation. Transformations between types of array and records are taken to determination of isomorphism between the main thing great numbers of the proper algebraic systems by the operations of change of structured level of data—selectors and constructing. For tract of land an operation of selector is taken to the reflection of set of indexes on the set of values of array cells. Such operation is like determined for record as a reflection between selectors of components and components.

Formal inequivalent type of data conversion in PL is executed by the next procedures.

1) Construction of operations of type of data conversion $T = \{T_\alpha^t\}$ for the set of programming *languages* $L = \{l_\alpha\}_{\alpha=1,n}$.

2) Construction of reflection of primary types of data for every pair of the connection modules in $l_{\alpha 1}$ and $l_{\alpha 2}$ and application of operations of the S selector and designer C for the reflection of complex data structures in these languages.

Formal type of data conversion is carried out for every type of T_α^t

$$T_\alpha^t : G_\alpha^t = \langle X_\alpha^t \Omega_\alpha^t \rangle$$

where t—type of data; X_α^t—the set of values, which can accept the variables of this type; Ω_α^t—the set of operations above these types of data.

For the primary and complex types of PL data of the algebraic systems are built:

$$\Sigma_1 = \{G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r\}$$
$$\Sigma_2 = \{G_\alpha^a, G_\alpha^z, G_\alpha^u, G_\alpha^e\}. \quad (5.2)$$

Every element of class of primary and complex types of data is determined on the set of their values and operations above them:

$$G_\alpha^t = \langle X_\alpha^t \Omega_\alpha^t \rangle, \text{ where } t = b, c, i, r, a, z, u, e.$$

The isomorphism reflection of two algebraic systems with the compatible types of data of two different PL corresponds to the operations of transformation of every t type of data. In class of the systems Σ_1 and Σ_2 types of data of conversion $t \to q$ for pair of languages of l_{et} and l_q next properties of reflections are definite:

1) G_α^t and G_β^q—isomorphism (q definite on that great number, that and t);

2) Between X_α^t and X_β^q exists isomorphism, for which set Ω_α^t and Ω_β^q different. If $\Omega = \Omega_\alpha^t \cup \Omega_\beta^q$ not emptily, consider isomorphism between $G_\alpha^{t'} = \langle X_\alpha^t \Omega \rangle$ and $G_\beta^{q'} = \langle X_\beta^q \Omega \rangle$. Such transformation is taken to the first case.

3) The powers of the algebraic systems must be equal $|G_\alpha^t| = |G_\beta^q|$.

Any reflection 1) 2) saves a linear order, because the algebraic systems (1) are linearly well-organized.

Lemma 1.1. For any isomorphism reflection φ between the algebraic systems G_α^t and G_β^q equalities are executed $\varphi(X_{\alpha\cdot\min}^t) = X_{\beta\cdot\min}^q \varphi(X_{\alpha\cdot\min}^t) = X_\beta^{q\max}$.

Formal terms of type of data of t conversion = b, c, i, r, a, z, u, e. are determined by the theorems 1 - 5 [10].

Proof of this theorem is banal and is investigation of properties of elements of the set.

Theorem 11. Let's φ—reflection of the algebraic system G_α^c in the G system $_\beta^c$. For that that φ was isomorphism's, it is necessary and enough, that φ isomorph mapping X_α^c on X_β^c with saving a linear order.

Necessity. Let's φ—isomorphism. Then in case of reflection all operations of set are saved $\Omega = \Omega_\alpha^c = \Omega_\beta^c$, including relational operator, which determines the linear order X_α^c and X_β^c.

Sufficient. Lets φ isomorphism mapping X_α^c on X_β^c—with saving a linear order. A relational operator is executed according to the efficiency principle. Will prove the operation of *succ* by lemma, in obedience to which equality is executed $\varphi(X_{\alpha\cdot\min}^c) = X_{\beta\cdot\min}^c$.

Consistently applying an operation of *succ* to this equality and taking into account the linear efficiency X_α^c and $X_\beta^c(x < succ(x))$, get, that for any $x_\alpha^c \in X_\alpha^c$ and $x_\alpha^c \neq X_{\alpha\min}^{c-}$ from equality $\varphi(X_\alpha^c) = x_\beta^c$, where $x_\beta^c \in X_\beta^c$, equality is executed

$$\varphi(succ(x_\alpha^c)) = succ(x_\beta^c). \quad (5.3)$$

An operation of pred is proved like with the help $\varphi(X_{\alpha\max}^c) = X_{\beta\max}^c$.

Theorem 12. Any isomorphism φ between the algebraic systems G_α^b and G_β^b is identical isomorphism:

$$\varphi\left(X_{\alpha.\text{falseb}}\right) = X_{\beta.\text{falseb}}$$
$$\varphi\left(X_{\alpha.\text{trueb}}\right) = X_{\beta.\text{trueb}}. \tag{5.4}$$

Proof. In case of the *G* reflection ${}^{b}_{\alpha}$ and G^{b}_{β} always justly $X_{\alpha.\text{ false } b} < X_{\beta.\text{ true } b}$. Therefore, taking into account saving a linear order, possible isomorphism solely is (3).

Theorem 1.3. Any isomorphism between the algebraic systems with the factual number types is identical av-toisomorphism.

Proof of this theorem is banal and is investigation of properties of elements of number great numbers.

Theorem 1.4. Lets G^{an}_{α} and G^{a}_{β}—algebraic systems, which answer the types of data of tract of land (*a*); φ_i and φ_v—isomorphism reflections of great numbers of indexes (*i*) and values of elements (Y) x0 of (Y) x0tracts of land, which save a linear order. Then isomorphism φ between the algebraic systems is wholly determined by the isomorphism reflections:

$$\varphi_i : X^{a}_{\alpha} \to X^{a}_{\beta}$$
$$\varphi_v : At\left(X^{a}_{\alpha}\right) \to AT\left(X^{a}_{\beta}\right). \tag{5.5}$$

Isomorphism φ between the algebraic systems G^{an}_{α} and G^{an}_{β} is determined by the reflections φ_i and φ_v, which are saved by the linear order and efficiency of array cells.

Theorem 1.5. Lets G^{z}_{α} and G^{z}_{β}—two algebraic systems, which answer the types of the data “record” or structure and $x^{z}_{\alpha} \in X^{z}_{\alpha}$, $x^{z}_{\beta} \in X^{z}_{\beta}$. Then, if between sequences of components of records of x^{z}_{α} and x^{z}_{β} exists mutually univocal correspondence, isomorphism φ between G^{z}_{α} and G^{z}_{β} is determined by the isomorphism reflections of the algebraic systems, to which the components of record or structure correspond.

Transformations between tracts of land and records are taken to the primary types of data of their elements conversion. The transformations between the actual types and other number values suppose the use of empiric cases, because isomorphism of the main thing great numbers of these algebraic systems is absent.

In case of primary and structural type’s conversion operations of the *S* selector and constructing are used *C* for the change of level of structural of data. An operation of the *S* selector for tract of land is determined as limitation of reflection:

$$M : I \to AT \text{ on } I', E \subset M : I \to AT \tag{5.6}$$

where *E*—the $I' \subset I$. then $M|\{k\}$ *k corresponds—array cell at* $I' = \{k\}$. This operation is like determined and for the *M* record|{Svm}, where *M*—reflection between selectors of components and components, and *Svm* determines the proper component of record.

The operation of constructing *from* tract of land consists of formal adduction in order of components and determination of accordance between the set of indexes and set of array cells. This operation is like determined for record.

Thus, the set of the operations *P*, *S* and *C* determines elementary rules for constructing complex types of data more from simple for the interactive modules on different to PL.

5.2. Use Assembly Method

A method of assembling based on the formal transformation of the different language modules was used during realization in the systems:

- APROP system was based on aggregate of the modules of Bank modules [1] [2];
- complex Program-Program (1987-1988) on the project KP NTP SEV-87 “INTERFACE-SEV” [3];
- APFORS system for automation of application packages creation on project of GKNT USSR (1988-1991) [4];
- Consulting model for information of users about the prepared modules of repeated use from Bank of the modules and selection of them for application in the practical purposes (1990).

An interface vehicle is used in paradigm of the programming.

5.3. Development of a Method of Interaction

Definition* 4.1. *Interoperability is the ability of components or systems to interact with each other and exchange common data.

Formally, interoperability model is understood as the representation of relationship parameters between different elements of software or informational systems. This model reflects the relationship system and the designing process for software product development. The relationships may be described with mathematical means, such as abstract algebras, standard OSI, interoperability theory and so on [9]-[19].

Interaction model is designed to share information between different components and systems in different environments. Interaction means links relations between two or more component objects. In the foundation of this interaction, there are messages sent by processes between applications, systems and environments. In the communications, the interface is defined in IDL or other language (APL, SIDL, etc.). In general, the interface is a mechanism for interoperability of applications for modern heterogeneous environments that support the development of applications and systems [14]-[16].

The interaction model describes relationship between the components and the application through the transmitted parameters. The model M_{inter} has the following representation:

$$M_{inter} = \left\{ M_{pro}, M_{sys}, M_{env} \right\}, \tag{5.7}$$

where $M_{pro} = \{C, Int, Pr\}$ is the application model, C is an object component, Int is an interface, Pr is the data transfer protocol; $M_{sys} = \{PS, Int, Pr\}$ is the system model; $M_{env} = \{Envir, Int, Pr\}$ is the environment model, where Int, Pr contain set of external interfaces and calls for data transfer between applications over network.

The basic parameters of the interaction model M_{inter} are application, interface and message.

Interative cooperation models are implemented in the ITC. Examples of pairs of systems include Visual Studio ↔ Eclipse, CORBA ↔ VS.Net, and IBM VSphere ↔ Eclipse. Their cooperation is secured by the specific mechanisms in the modern system environments [17]-[19].

Implementation of the Interaction model

The applied models for system interconnection via Eclipse IDE have the following implementation in the ITC [13] [14]:

Visual Studio.NET, Eclipse implement interaction of software systems in C# and Java programming languages according to the description of their interfaces and stubs/skeletons, which help restructure transmitted data.

CORBA, Java, MS.Net support connections between heterogeneous programs in the Eclipse repository and processing the transmitted data.

IBM VSphere, Eclipse support merging software systems built from heterogeneous components and two-way data transmission. The basic parameters for the interoperability model M_{inter} are program, interface and message or protocol.

6. Technologies of Assembling Elements of Paradigms

Methodology of planning and realization by the AS method of the assembling programing from the *prepared elements of paradigms* by CASE-instruments (translators with PL, testing, transformation, generators and etc.), and also instrumental facilities (Eclipse, Protege, VS.Net, Corba, Java and т. п) are realized in ITC [6]-[17].

In ITC a prepared resource of paradigms is been by RC (reuse, assets, artifacts, services and etc.). They reflection some functions of the functional objects. Each RC is specified by the proper standards by specification of realized functional elements in WSDL (**Figure 7**) and interface in the languages IDL, API, SDIL and so on. It gives possibility to put together RC on the single basis, general for all types of heterogeneous resources [18].

Principles of development of the systems from the prepared program and informative by ready resources (components, services, interfaces, data, artifacts and etc.) such:

- composition of the complex systems of the distributed AS type from components, interfaces and services with their properties, descriptions and machineries of aggregation in the more complex structures and rules of co-operation in the integrated environments;
- component engineering (CBSE), as distributed AS creation from the prepared "details" activity, based on the really existent positions of engineering products, namely, is standard the Cycle Life requirements, exploitation and RC or SPF elimination with the use of the systems of classification and the RC cataloguing, standardization facilities, standardization of RC description and integration of them in AS and SPF;
- Interoperability RC and the distributed AS elements, which is based on interfaces and rules of co-operation

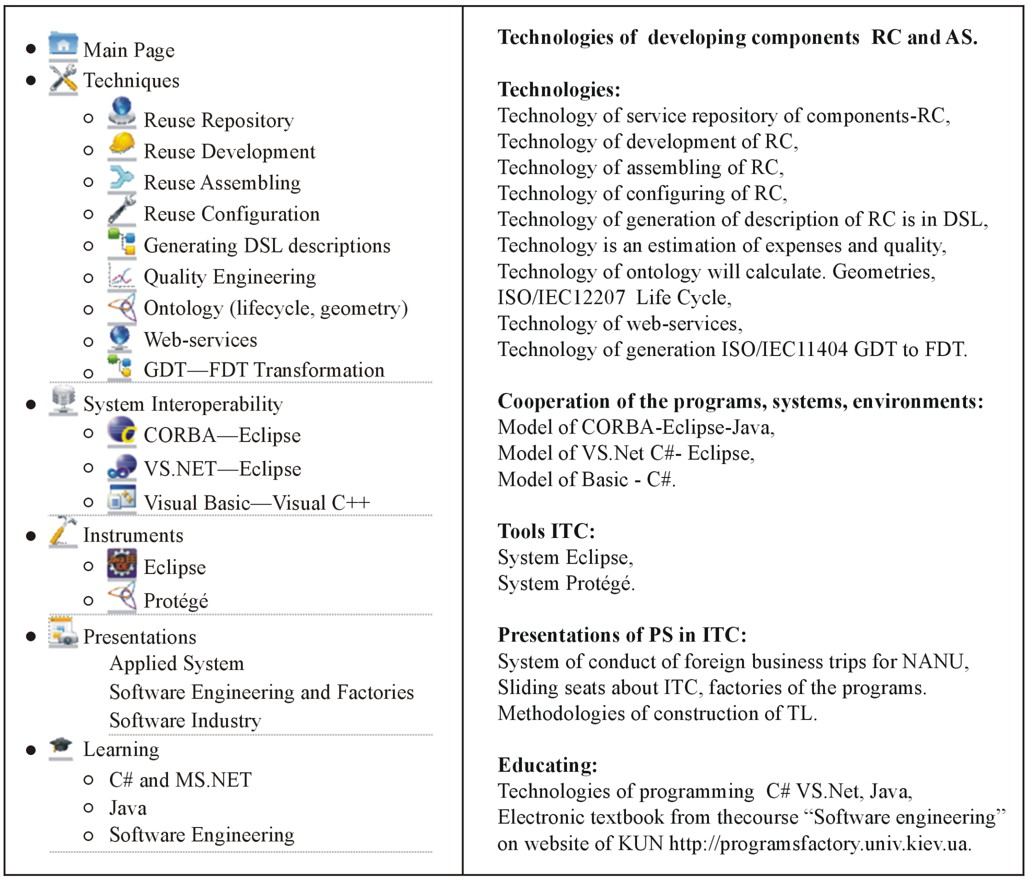
of components between itself for providing integration and functioning in the different environments;

Variant as power of RC and component AS to the changes by elimination of some of functional, uncompleted either additions of new functional RC in the configuration structure SPF or distributed AS and etc.

The technology of the PS planning from RC includes (**Figure 8**):

- the PS planning with the use of the models MDD, MDA and processes of standard CL;
- Ontological planning of domains with the task of model of descriptions and system architecture from the prepared components;
- Specification of heterogeneous program resources in ЯП, their realization, verification of their rightness by verification and testing a component in the tasks of passport of prepared element;
- Selection functionally prepared components in repository;
- Assembling heterogeneous components of repeated use RC and passed between them data, irrelevant on type, format, size and so on. Conversion;
- From some artifacts of source code and adaptation of them under the concrete purposes of before created program decision or program;
- Description of specific about by the DSL language facilities with the use of DSL Tools' VS.Net for the receipt of output code;
- RC and PS testing with the necessary data capture for the PS quality estimation;
- Engineering quality of the program systems, including an expert and metrical analysis of indexes of quality and achievement of product quality indexes;
- Saving results of planning in repository of components;
- RC documenting, new functional components and so on.

Elements of paradigms of programming, necessary at planning domains are realized in the ITC complex, PS from RC. In him found the reflection fundamental positions of led paradigms of programming, including; theory



- Main Page
- Techniques
  - Reuse Repository
  - Reuse Development
  - Reuse Assembling
  - Reuse Configuration
  - Generating DSL descriptions
  - Quality Engineering
  - Ontology (lifecycle, geometry)
  - Web-services
  - GDT—FDT Transformation
- System Interoperability
  - CORBA—Eclipse
  - VS.NET—Eclipse
  - Visual Basic—Visual C++
- Instruments
  - Eclipse
  - Protégé
- Presentations
  - Applied System
  - Software Engineering and Factories
  - Software Industry
- Learning
  - C# and MS.NET
  - Java
  - Software Engineering

Technologies of developing components RC and AS.

Technologies:
Technology of service repository of components-RC,
Technology of development of RC,
Technology of assembling of RC,
Technology of configuring of RC,
Technology of generation of description of RC is in DSL,
Technology is an estimation of expenses and quality,
Technology of ontology will calculate. Geometries,
ISO/IEC12207 Life Cycle,
Technology of web-services,
Technology of generation ISO/IEC11404 GDT to FDT.

Cooperation of the programs, systems, environments:
Model of CORBA-Eclipse-Java,
Model of VS.Net C#- Eclipse,
Model of Basic - C#.

Tools ITC:
System Eclipse,
System Protégé.

Presentations of PS in ITC:
System of conduct of foreign business trips for NANU,
Sliding seats about ITC, factories of the programs.
Methodologies of construction of TL.

Educating:
Technologies of programming C# VS.Net, Java,
Electronic textbook from thecourse "Software engineering" on website of KUN http://programsfactory.univ.kiev.ua.

Figure 8. Keywords in the main page of the web site ITC.

of co-operation and the PS variant; theory of design and adaptation of projected AS in other environments, technology of the ITC making on the 10 lines from RC; Cycle Life 12207 standard and ISO/IEC 3226 quality estimation processes, ontology of calculable geometry; line of teaching to the modern languages C#, Java and CASE-instruments—Protégé, Eclipse, VS.Net, Java and so on.

To the technological lines it is possible to apply through web-site (http://7dragons.ru/ru). For teaching to the object the "Program engineering" it is possible to apply to e-textbook author on the experimental factory of the programs http://programsfactory.univ.kiev.ua) in Ukrainian and to web-site www.intuit.ru in Russian.

7. Conclusion

The theory and paradigms assembling of producing PS from reusable software resources was developed over many years and entered the practice of reuses in many countries. In a single conceptual framework, this technology combines theoretical object simulation and OM with object-functions and interfaces. At the same time, a large number of software reusable resources have been accumulated along with the development of object and components by interfaces communication. This new approach allows constructing compound applications by using various kinds of systems, functional components and services. Some aspects of the theory focused on the interoperability of software resources and configuring these resources in PS. Considered programming paradigm taught in the KNY and the MIPT, as well as the students developed individual thesis on this topic.

References

[1] Lavrischeva, E. and Grischenko, V. (1982) The Connection of Multi-Language Modules in OS ES, M.: 137 p.

[2] Lavrischeva, E. and Grischenko, V. (1991) Assembly Programming. K: Nauk.dumka, 213 c.

[3] Lavrischeva, E. and Grischenko, V. (2009) Assembly Programming. Basics of Software Industry. 2nd Edition, Naukova Dumka, Kiev, 371 p. (In Russian) http://www.twirpx.com

[4] Lavrischeva, E. (2008) Software Engineering (in Ukrainian). Akademperiodika, Kiev, 319 p.

[5] Lavrischeva, E., Koval, G., Babenko, L., Slabospitska, O. and Ignatenko, P. (2011) New Theoretical Foundations of Production Methods of Software Systems in Generative Programming Context. IK-2011, Software Institute NANY, 277 p. http://www.nbuv.gov.ua/

[6] Lavrischeva, E. (2008) Formation and Development of the Modular-Component Software Engineering in Ukraine. Akademperiodika, Kiev, 31 p. (In Russian)

[7] Lavrischeva, E. and Petruchin, V. (2007) Methods and Means of Software Engineering. 415 p. (In Russian) http://www.intuit.ru/ and http://www.twirpx.com/

[8] Lavrischeva, E. (2006) Methods Programming. Theory, Engineering, Practice. Naukova Dumka, Kiev, 451 p. (in Russian)

[9] Lavrischeva, E.M. (2014) Software Engineering Computer Systems. Paradigms, Technologies, CASE-Tools. Naukova Dumka, Kiev, 284 p. (In Russian)

[10] Lavrischeva, E., Stenyashin, A. and Kolesnyk, A. (2014) Object-Component Development of Application and Systems. Theory and Practice. *Journal of Software Engineering and Applications*, **7**, 756-769. http://dx.doi.org/10.4236/jsea.2014.79070

[11] Lavrischeva, E. and Ostrovski, A. (2013) New Theoretical Aspects of Software Engineering for Development Applications and E-Learning. *Journal of Software Engineering and Applications*, **6**, 34-40. http://dx.doi.org/10.4236/jsea.2013.69A004

[12] Radetskyi, I. (2011) One of Approaches to Maintenance Interconnection Environments Visual Studio and Eclipse. *Problems in Programming*, **2**, 45-52. (In Ukrainian)

[13] Lavrischeva, E., Dzubenko, A. and Aronov, A. (2013) Conception of Programs Factory for Representation and E-Learning Disciplines of Software Engineering. 9th International Conference ICTERI, ICT in Education, Research and Industrial Applications, Integration, Harmonization and Knowledge Transfer, Ukraine, 17-21 June. http://ceur-ws.org/Vol-1000/

[14] Aronov, A. and Dzubenko, A. (2011) Approach to Development of the Students' Program Factory. *Problems in Programming*, **3**, 42-49. (In Ukrainian) http://www.isofts.kiev.ua/

[15] Grischenko, V. (2007) Object-Component Designing Method for Software Systems. *Problems in Programming*, **2**, 113-125. Akademperiodika, Kiev. (In Ukrainian)

[16] Lavrischeva, K. (2010) Formal Fundamentals of Component Interoperability in Programming. *Cybernetics and Systems Analysis*, **46**, 639-652. Springer, Heidelberg. http://dx.doi.org/10.1007/s10559-010-9240-z

[17] Lavrischeva, E., Zinkovich, V., Kolesnik, A., *et al*. (2012) Instrumental and Technological Complex for Development and Learning Design Patterns of Software Systems. State Intellectual Property Service of Ukraine, Copyright Registration Certificate No. 45292, 103 p. (In Ukrainian)

[18] Lavrischeva, K.M. (2012) Component Programming. Theory and Practice. *Problems in Programming*, **4**, 3-12. (In Ukrainian) www.isofts.kiev.ua

[19] Lavrischeva, K.M. (2011) Interaction of Programs, Systems and Operating Environments. *Problems Programming*, No. 3, 11-23.