IEEE Access
practical innovations : open solutions

Received May 13, 2013, accepted June 26, 2013, date of publication July 25, 2013, date of current version August 6, 2013.

*Digital Object Identifier 10.1109/ACCESS.2013.2274896*

# A Scientifically Experimental Approach to the Simulation of Designer Activity in the Conceptual Designing of Software Intensive Systems

**PETR SOSNIN (Member, IEEE)**
Computer Department, Ulyanovsk State Technical University, Ulyanovsk 432027, Russia (sosnin@ulstu.ru).

**ABSTRACT** The success of designing software intensive systems (SISs) may be improved by incorporating experimentation to be part of the design process. This paper presents a scientific approach to experimentation on objects that are units of designers' behavior and is aimed at solving project tasks in conceptual design. The proposed approach is based on specifying the behavior units as precedents and pseudo-code programming of experiments' plans. Reasoning used by designers in the experiments is registered in a question-answer form. Experimenting is supported by a specialized toolkit.

**INDEX TERMS** Conceptual designing, designers' behavior, experience, question-answering, precedent, software intensive system.

## I. INTRODUCTION

In reality, designing SISs has been associated with a very low degree of a success (approximately 35%) for years [1]. Such a state of affairs is caused basically by the insufficiency of reliable ways-of-working used by a team of designers under the conditions of a complicated computerized environment. Insufficient reliability is a consequence of many problems, the main source of which is the existing experience of software engineering in the context of its application to the design of SISs.

A promising attempt at positive re-founding of software engineering is bound with an SEMAT initiative (Software Engineering Methods And Theory initiative) in documents for which a way-of-working used by designers is marked as a very important essence [2]. There, ''way-of-working'' is defined as ''the tailored set of practices and tools used by a team to guide and support their work.''

The description above indicates that ways-of-working are very important objects of scientific investigations aimed at predicting results. It is our deep belief that investigations of this type should be fulfilled not only for the subject area of designing the SISs as a whole but in every project that involves SISs as well. In the design of a specific SIS, the research of the used way-of-working should be necessarily fulfilled by the team of designers.

It is necessary to clarify the proposed position. Way-of-working, when used in the design of a definitive SIS, should be created step-by-step, and the steps should include research conducted by a designer or by a designers' group in the form of a scientific experiment. If such experiments are aimed at simplifying the complexity of the collaborative work of the designers, then their inclusion to way-of-working will facilitate achieving success in the design of SIS.

In general, the complexity (or simplicity) of SIS reflects the degree of difficulty for designers in their interactions with specific models of SIS (or its components) in solving specific tasks. The system or its components are complex if the designer (interacting with the system) does not have sufficient resources to achieve the necessary level of understanding or to achieve the other planned aims.

Often, various interpretations of the Kolmogorov measure [3] are applied for estimations of the degree of the system's complexity. This measure is connected with the minimal length of program P that provides the construction of system S from its initial description D. Distinctions in interpretations are usually caused by features of the system S and formal descriptions to be used for objects P and D.

In accordance with their destinations, for the programs of P-type, it is more preferable to build them as programmable systems of designers' actions while using the specific methods of meta-programming M. Explicit and/or implicit methods for managing the designers' activity are used in the technology behind SIS design. The reality of such management demonstrates that the complexity of a P-program is no

less than the complexity of SIS in any of its used states. Moreover, any M-program that provides the construction of a P-program should be built on the basis of the same initial description D of the system S. It can be presented by the following chain: $D(t_0) \rightarrow M(t_i) \rightarrow P(t_j) \rightarrow S(t_n)$.

Named relations between constructions D, M, and P can be used by designers to divide the process of designing into stages $[D(t_0) \rightarrow M_0 \rightarrow P_0 \rightarrow S(t_1)]$, $[D(t_1) \rightarrow M_1 \rightarrow P_1 \rightarrow S(t_2)]$, ..., $[D(t_i) \rightarrow M_i \rightarrow P_i \rightarrow S(t_{i+1})]$, ..., $[D(t_{n-1}) \rightarrow M_{n-1} \rightarrow P_{n-1} \rightarrow S(t_n)]$, where a set $\{S(t_i)\}$ reflects the states of the SIS that are being created.

Dividing the designing process into stages is a typical step in any modern technology that involves the creation of SISs. In different technologies, such an approach is used in different forms for different aims. This approach helps to decrease the complexity of the interactions with SIS in any of its states S(ti). However, until now, the viewpoint of programming on the designer activity has not been supported instrumentally in the early design stages.

It is necessary to note that the creation of instrumental means for an explicit work with M- and P-type artifacts essentially depends on their understanding. In an approach described in this paper, these artifacts are understood as models of designers' behavior that are created for scientific experimentation on the corresponding behavior units included in the way-of-working. Therefore, the indicated instruments should have the potential for the creation and use of the artifacts of M- and P-types in experiments with behavioral units of designers' activity. Moreover, the means of supporting the experimental work should help the designers create solutions for investigating tasks and opening the possibility for their confirmatory reuse in the designers' team.

The specificity of the approach is defined by the following features.

1. The investigated units of behavior are interpreted as precedents, with which the designers interact, using accessible experience and its models.
2. Experimenting with the chosen unit of behavior, the designer creates the corresponding precedent model that fulfills the function of ''experimental setup.''
3. Any such ''experimental setup'' is built to confirm the existence of a specific ''cause-and-effect regularity (or regularities)'' in the ''naturally artificial world'' of the SIS that is being created.
4. The existence of any investigated ''cause-and-effect regularity'' should be confirmed not only by an author of the experiment but also by other members of the designers' team.
5. Interactions of designers with the experience and its models (i.e., the accessible experience) are based on question-answer reasoning.
6. The approach is aimed at the conceptual stage of designing because a system of SIS regularities should be formed, checked and confirmed during the design as early as possible.
7. Artifacts of M- and P-types (which are included in models of precedents) are programmed in a specialized pseudo-code language that is used by designers in coordination with question-answer reasoning.

In the proposed approach, the achievement of the enumerated features is provided by a specialized toolkit WIQA (Working In Questions and Answers) [4], which supports the process of experimenting during the stages of preparing, conducting and registering the experiments in their understandable forms.

## II. PRELIMINARY BASES

### A. WHY EXPERIMENT WITH ACTIVITY UNITS OF DESIGNERS

In a general case, a system of the SIS-type involves software that is combined with peopleware and other different components. Any such system can be interpreted as a naturally artificial world, the processes of which are implemented in accordance with specific cause-and-effect laws (regularities). Some of these regularities are laws of nature, while others have a normative character.

In designing an SIS and its cause-and-effect regularities (or shortly regularities), the designer must find a reflection of the design in the software components. In software engineering, the reflected regularities are bound with requirements and restrictions that are embodied in the created software. Such embodiments are implemented when designers formulate and solve the corresponding project tasks. Embodiments of requirements and their subsets are distributed between project tasks. It is necessary to note that this part of way-of-working is based on the experiences of requirements engineering.

Thus, in a general case, the project task concerns a set of specific regularities, the coexistence of which should be confirmed for SIS, which otherwise is not constructed. Under such conditions, the necessary confirmation can be implemented by the designer who performs the appropriate actions, with a scheme for the corresponding part of the SIS.

Thus, acting in accordance with the created scheme, the designer should be convinced that the scheme of actions leads to the necessary use of cause-and-effect regularities in the task. Moreover, for mutual understanding and collaborative work, the designer should specify the scheme for reuse by other members of the team. A similar responsibility is expected of a scientist who has solved an experimental task.

Therefore, simulating the scientists' activity in experimenting with activity units of designers is a promising way of improving the ways-of-working when designing the SISs. This position focuses on the system of cause-and-effect regularities, which manages the processes in the naturally artificial world of SIS; it correlates with the empirical nature of software engineering.

### B. WHY CONCEPTUAL DESIGN

There were several reasons for the choice of conceptual design as an activity domain for the constructive use of the analogy between designing and scientific experimental research. The first reason is connected with the high cost of

errors that mainly arise from incorrect understanding of the cases that the designers are working on.

Any conceptual project is aimed at a description of the system that is being created, to allow for its structure and behavior to be coordinated with natural laws and normative rules the system should satisfy. Hence, designers should prove that the used models of ''cause-and-effect regularities'' provide the required coordination. This approach should be put into practice in the conceptual design of a specific SIS as early as possible. The aim of any scientific experiment is an existence of the confirmation of a corresponding cause-and-effect regularity. The similarity of the obligations of designers and scientists was the second reason for this choice.

The third reason is the lack of methods that are included, especially modern technologies for supporting the experimental activity of designers at the conceptual stage of design. It is necessary to note that, at the conceptual stage of design, the investigated scheme of the designer's actions can be a simplified version of the task solution, demonstrating only how regularities of the task can be materialized and will be used.

### C. WHY DESIGNERS WORK IN THE FORM OF PRECEDENT

In the offered approach, this work with regularities is considered from the viewpoint of the designers' behavior in the solution processes of the project tasks. Moreover, the approach is oriented toward the analogy between projects completed by designers and scientists. In specific circumstances, any designer plays the role of a scientist who prepares and conducts experiments with P-type behavior units. In such experiments, the designer works in the naturally artificial world of SIS, which is developed in the technological medium used. Any experiment is connected with solving the corresponding task appointed to the designer on the team.

When solving the appointed task, the designer must focus on the causes-and-effects of this task, to prepare an experiment that confirms that the indicated causes-and-effects exist in the investigated world. In the experiments, the designers should focus on ''how to do'' but not on ''what to build,'' which correlates with the main principles of SEMAT. In experimenting, the designers acquire units of experience, which they should register in the form of models that provide future reuses of the experiments. It is necessary to notice that typical (reused) units of human behavior are usually called ''precedents.''

According to the Cambridge dictionary, ''precedents are actions or decisions that have already happened in the past and which can be referred to and justified as an example that can be followed when the similar situation arises'' (http://dictionary.cambridge.org/dictionary/british/precedent).

Natural precedents are based on conditioned reflexes, which are intellectually processed and are included as models of the human experience, as its units. An interpretation of behavior units as precedents prompts the necessity of intellectual processing them, which is aimed towards their planned reuse. In the offered approach, the intellectual processing is based on the logical scheme of precedents presented in Fig. 1.

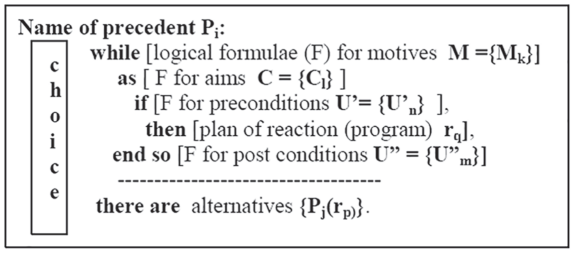


**FIGURE 1.** Logical scheme of precedents.

The applied logical scheme allows integrating the natural and normative regularities in the precedent model. Natural regularities are reflected in pre- and post-conditions, while normative regularities are specified in the motives and aims components.

### D. WHY INTERACTIONS WITH EXPERIENCE

This work of the designer as a type of an activity is initially oriented to the use of experiential practices. Any new SIS project is evolved through the real-time solution of the technological and subject tasks that are distributed between members of the team [5]. This distribution accounts for the personal experience of any member with real-time access to the collective experience and its models.

It is easy to agree that the contribution of any designer to the general work depends on an essential measure both from the accessible experience and from the used means of the access. With any newly solved task, any designer evolves their own experience, further enhancing the current state of the common accessible experience. Such a process is similar to experiential learning [6] under the conditions of designing a specific SIS. The most important aspect in experiential learning is the fact that the learner (in our case, the designer) should actively participate in purposeful actions and reflect them when using the accessible experience.

Thus, one can interpret the designing of SISs from the viewpoint of experiential learning in the team of designers. This viewpoint indicates an important area for rationalizing (improving) the real-time actions of designers aimed at interactions with the accessible experience. Rational workflows of such interactions can increase the degree of success in the development of SISs. The continuous improvement of the used practices is recognized as a reliable way of achieving sustainable success of the team of designers and their project organization [7].

### E. WHY QUESTION-ANSWERING

Intellectual processing of natural precedents is fulfilled by means of a natural language that also supports the access to the units of experience. Such effects are caused by processes

of consciousness of a dialog nature. Dialogical processes of consciousness provide a choice of experience units and their adjustments for corresponding situations.

Accordingly, a question is understood by the author as the natural phenomenon associated with a specific situation in which a human interacts with their own experience (models of precedents). In this case, the ''question'' is a symbolic (sign) model of the corresponding question. The associated understanding helps explain the necessity of fitting the ''question'' to the investigated situation. Implicit questions and answers exist in actuality, while ''questions'' and ''answers'' are presented as symbolic models.

Thus, question-answer reasoning reflects processes in the consciousnesses of designers when they interact with their own experience. Explicit question-answer reasoning helps in controlling interactions with experience. This type of reasoning is expedient for the creation and use of experience models that present the designer's behavior.

### F. WHY PSEUDO-CODE PROGRAMMING

In activity practice, the person usually represents their own behavior by means of a plan that is written in natural language based on its algorithmic usage. Repeatable work that is fulfilled by people is represented by techniques that are also written in natural language based on its algorithmic usage. This approach prompts the designer to use this technique in planning experiments, which was the reason behind choosing a pseudo-code language for programming the units of the designer's ideas. Interacting with such programmed models of behavior, the designers will apply their experience in the use of natural language.

The logical scheme of the precedent specifies two parts of its model because of their separate pseudo-code programming. The first part must define the access to the precedent model, while the second part must describe the corresponding reaction that must be implemented by the designer.

It is necessary to recall the M- and P-types of programs that were chosen to describe the designers' activity in the offered approach. Let us clarify the difference between these types of programs. P-programs are destined for simulating the units of the designer's behavior in the solutions of the appointed tasks, while M-programs describe the collaborative work of the designers. Thus, M-programs are destined for controlling the activity of the designers, while they work in parallel and in coordination. This type of program is also suitable for controlling the pseudo-parallel work of the designer, with a number of appointed tasks. In general, the M-programs automate the implementation of the workflows.

Workflows in conceptual designing are formed according to the planned schemes; thus, they conform with the current situation. Workflows of the planned type can be programmed in advance, but workflows of the second type are to be programmed by working designers (not professional programmers) in real time. Hence, the language of pseudo-code programming, which will be used by designers for the creation of P-programs, can be used for the creation of M-programs also.

### G. PLACE OF OFFERED MEANS IN THE CREATION OF WAYS-OF-WORKING

In accordance with its definition, the way-of-working includes the tailored set of practices chosen for the SIS design. The presented approach concerns the used practices, and mechanisms of their tailoring applies to the processes of conceptual designing. The specificity of the approach will be clarified by means of roles that provide the constructive work of designers in technological environments.

Any modern technology that is used for designing the SIS includes modeling this work of designers with the help of roles. For example, the current version of Rational Unified Process (RUP) supports the activities of designers playing approximately 40 roles. RUP is a ''heavy'' technology; therefore, for small teams, the quantity and specifications of the used roles are decreased and simplified [8].

In any case, the role is a special version of a designer's behavior that satisfies a certain set of rules. Role specifications depend on the corresponding tasks and tools that support their initial solution and reuse of solutions. Thus, for specifications of any role, the corresponding tasks and necessary tools should be defined by their essential features.

This approach recommends expanding the traditional set of roles by including into this set the specialized role called ''intellectual processor'' (I-processor). This role exists in addition to any other role that is played by any designer in this study for the investigated task.

Acting as an I-processor, the designer constructively uses the accessible experience for different useful purposes. Interactions with experience units are implemented explicitly and/or implicitly but on the basis of QA-reasoning. The main purpose of using the explicit forms is the programmable creation of the necessary precedents and experimentation with them. Implicit forms are suitable for (pseudo-code) programming of used practices oriented to execution of their programs by a designer playing the role of I-processor. Thus, the applied way-of-working can be adjusted to the use of I-processors in situations where it is planned or estimated as being useful.

Actions of I-processors are supported by the toolkit WIQA, which provides the collaborative execution of workflows ''Interactions with Experience'' by the group of designers in the client-server medium [9]. The basic features of an I-processor will be specified below.

We note that a set of practices embedded into the workflows ''Interactions with Experience'' can be combined with other practices of the technology used.

## III. RELATED STUDIES

The idea of the designer model as an I-processor is derived from publications [10] in which the ''Model Human Processor'' (MH-processor) is described as an engineering model of human performance in solving different tasks in a real-time regime. It is necessary to emphasize that I-processor is

similar to MH-processor and includes the cognitive processor. However, the existence and work of this ''component'' of I-processor are derived from another technique that is based on reasoning of the question-answer type.

The EPIC version of MH-processor uses programs that are written in the specialized command language Keystroke Level Model (KLM). A set of basic KLM actions includes the following operators: K – key press and release (keyboard), P – point the mouse to an object on screen, B – button press or release (mouse), H – hand from keyboard to mouse, or vice versa, and other commands. Operators of the KLM-language and their values are used to estimate temporal characteristics of human interactions for alternative schemes of interfaces. KLM-programs do not correspond exactly to the used reasoning, and therefore, they do not reflect interactions with the accessible experience. It is necessary to note that processors of both indicated types are oriented toward experimenting with units of human behavior.

The version of experimentation described is coordinated with basic principles of the SEMAT Kernel, which is described in [2]. The version is oriented towards ways-of-working that focus on the real-time activity of designers. ''The process is what the team does. Such processes can be formed dynamically from appropriate practices in accordance with current situations. The informational cards and queue mechanisms are being used for managing of ways-of-working [11].''

It is necessary to note that this version applies units of task types for structuring the activity of the designers because solutions of tasks facilitate the enrichment of the accessible experience by scientific experimentation. By experimenting, the scientists solve specific tasks creatively.

In the offered approach, the scientific viewpoint correlates with two faces of the software engineering described in [12], where functional paradigms and scientific paradigms are discussed. In the context of this paper, the approach means are oriented towards scientific paradigms used by software engineers.

Therefore, the important group of related studies includes publications that present empirical viewpoints on software engineering. In this group, we note the following works [13] and [14], which present the domain of empirical software engineering; papers [15] and [16], which define the Goal-Question-Metrics method (GQM-method) and Experience factory, which includes the Experience Base. All of the indicated studies were taken into account in the offered version of scientifically experimental ways-of-working.

One more group of related publications concerns the use of question-answering in computerized mediums, for example, papers [17] and [18]. In this group, the closest research presents experience-based methodology ''BORE'' [19], in which question-answering is applied as well, but for the other aims, this methodology does not support programming of the creative designer activity.

Explicit programmable forms of the designer activity are not used in any modern technologies of designing the SIS. For example, in technologies that are based on the Rational Unified Process [20], the conformity to requirements and understandability are reached with the help of ''block and line'' diagrams expressed in the Unified Modeling Language (UML). The content of the diagrams built by the designers is clarified by necessary textual descriptions. However, UML is not a language of the executable type, and therefore, the associated diagrams are not suitable for experimentation, as is the case with P-type programs.

For collaborative solution of the tasks, using coordination, the RUP suggests a means of using normative workflows in the relations that are regulated by a set of rules. For any task in a specific normative workflow, the RUP has an interactive diagrammatic model using a set of components that can help in solving the task. Forms of programming are not always used in these methods. A similar state of affairs with conceptual designing exists in other known technologies that support the development of SIS.

It should be noted that formal languages are widely used in applications that involve workflows. Therefore, a separate group of related research includes publications that open solutions that are connected with the use of workflows in collaborative designing. In this group, we note publications [21] and [22], which are helpful for gathering and defining a set of tasks that are used in the workflows ''Interactions with experience.''

## IV. QUESTION-ANSWER INTERACTIONS WITH TASKS

### A. OPERATING SPACE OF EXPERIMENTATION

Any experimental research is implemented in an appropriate medium of experimenting. In the described case, the role of such a medium fulfills an operating space that is supported by the toolkit WIQA. The generalized scheme of experimentation in the indicated space is presented in Fig. 2.

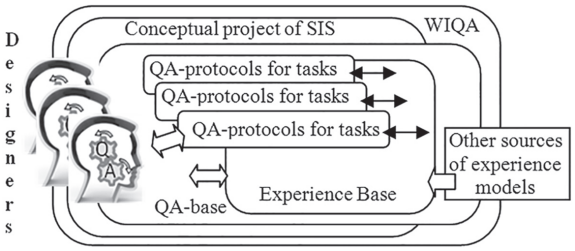


**FIGURE 2.** Experiential interactions with tasks.

In solving an appointed task, the designer registers the used question-answer reasoning (QA-reasoning) in a specialized protocol (QA-protocol) so that this QA-protocol can be used as the task model (QA-model). Typical units of QA-reasoning are questions (Q) and answers (A) of different types. Tasks are a very important type of question. Below, the tasks will be designated with the use of the symbol ''Z.''

Models of this type can be used by designers for experimenting in real time with all of the tasks being solved. Units of the experiential behavior that are extracted from the

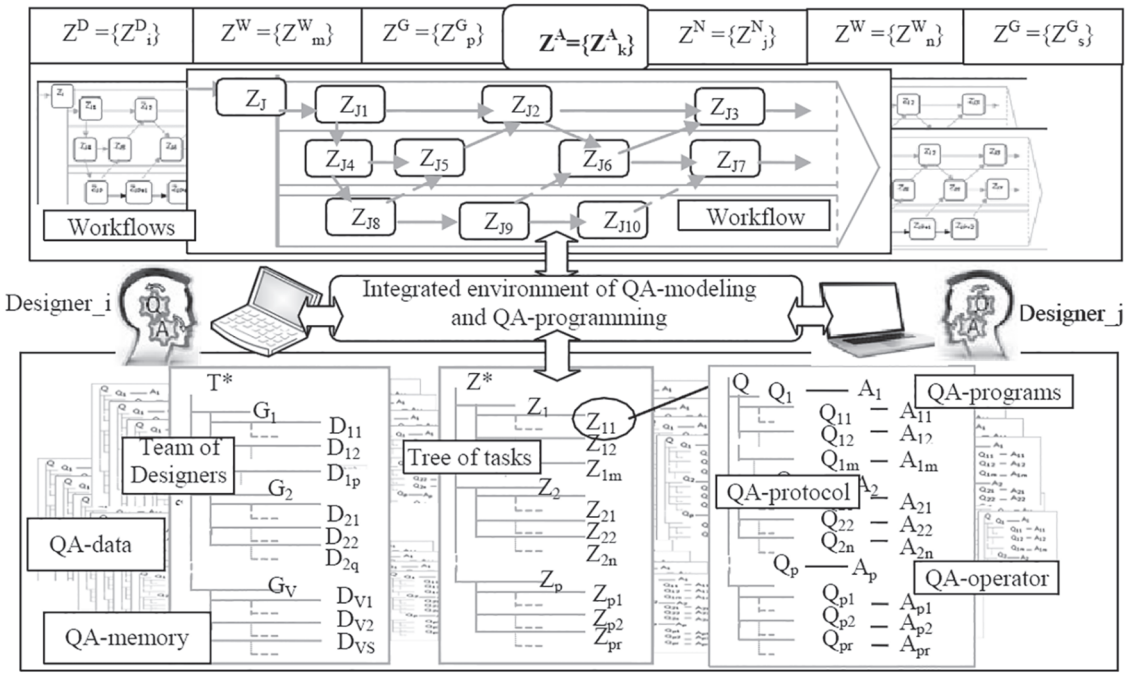


**FIGURE 3.** Storing of tasks in WIQA-environment.

solution processes are modeled on the basis of QA-models of tasks.

The scheme also reflects that the investigated behavior model can be uploaded as the model of the corresponding precedent in the question-answer database (QA-base) and in the Experience Base of WIQA. After that, they can be used by designers as units of the accessible experience. Experience models from the other sources can be uploaded in the Experience Base.

### B. QUESTION-ANSWER MEMORY

If designers of SIS use the toolkit WIQA, then they have the opportunity for conceptually modeling tasks of different types. In this case, the current state of the tasks being solved collaboratively is registered in the QA-base of the toolkit, and this state is visually accessible in the forms of a tree of tasks and QA-models for the corresponding tasks. The named opportunity is presented figuratively in Fig. 3, where QA-base is interpreted as a specialized QA-memory in which the cells are visualized by inquiries of designers.

First, the cells are used for storing the registered units of QA-reasoning. Any cell has the following basic features.

1. A cell is specified by a set of normative attributes that reflect, for example, the textual description of the stored interactive object, its type and unique name, the name of its creator, the time of the last modification and the other characteristics.
2. Any cell has a unique address, which has a function that is fulfilled by the type name of the stored unit and its unique index, which is appointed automatically when creating the unit. Empty cells are absent.
3. The designer has a chance to appoint a number of additional attributes to the cell if it would be useful for this work involving the object stored in the cell.

Having chosen the necessary attributes, the designer can adjust the cell for storing any question or any answer in the form of an interactive object that is accessible by inquiries of designers of programs. Thus, any question and its answer are stored in QA-memory as a pair of related interactive objects, which is called a QA-unit.

QA-units are stored in QA-memory as data; the abstract type data will be called QA-data. The use of this type of data helps to emulate other data types, including descriptions of operators. First, this capability is necessary for the use of QA-memory in pseudo-code programming. Thus, cells of QA-memory that are destined for storing QA-units can be adjusted to store other types of units, for example, for units used in solving the tasks.

In Fig. 3, the scheme of QA-memory demonstrates the store of presentations for ''Team of designers,'' ''Tree of tasks'' and a pseudo-code program with its operators and data. The program, its operators and used data are designated as QA-program, QA-operators and QA-data, respectively, to underline that they inherit the features of QA-memory cells.

The responsibility for the tasks being solved is being distributed among designers in accordance with the competence of each individual. The team competence should be sufficient for real-time work with the following sets of tasks: subject tasks $Z^S = \{Z_i^S\}$ of the SIS subject area; normative tasks $Z^N = \{Z_j^N\}$ of the technology used by designers; adaptation tasks $Z^A = \{Z_k^A\}$ that provide an adjustment of the

tasks $\{Z_j^N\}$ for solving the tasks $\{Z_i^S\}$; workflow tasks $\{Z_m^W\}$ that provide work with the tasks of $Z^S$-type in workflows $\{W_m\}$ in SIS; workflow tasks $\{Z_n^W\}$ that provide work with $Z^N$-type tasks in corresponding workflows $\{W_n\}$ in the used technology; and workflow tasks $\{Z_p^G\}$ and $\{Z_r^G\}$, any of which correspond to the specific group of workflows in SIS or in the technology.

The indicated diversity of tasks emphasizes that designers should be very qualified specialists in the technology domain but that this qualification is not sufficient for successful designing. Normative tasks are invariant to the SIS domain and, therefore, designers should gain certain experiences that are needed for solving the specific tasks in the SIS subject area. The largest part of the additional experience that must be acquired by designers is experiential learning when $Z^S$-type tasks are solved during conceptual designing. The solution of any task $Z_i^S$ is similar to expanding it into a series on the basis of normative tasks.

Objects uploaded to QA-memory are bound in hierarchical structures. In real-time work, the designers interact with such objects. They process them with the help of appropriate operation to find and test the solutions to the tasks.

Objects in QA-memory are accessible to designers in accordance with the given access rights. In any case, any QA-model is accessible to the group of designers who interact with it with different purposes, which include checking the model. Thus, any QA-model is a product of collaborative reasoning and coordinated understanding.

### C. QUESTION-ANSWER MODELING

One method for the conceptual solution of any task of the indicated types is based on creating its QA-model as a system of questions and answers that accompany the solution process. The generalized scheme of such a model is presented in Fig. 4.

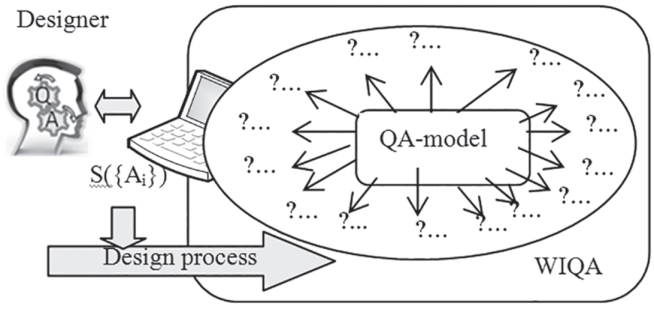


**FIGURE 4.** Interactions with QA-model of Task.

Question-answer and other models are created to make an extraction of answers to the questions enclosed in the model. Moreover, the model is a very important form of the representation of the questions, the answers of which are generated during visual interactions of designers with the model.

The essence of QA-modeling is the interactions of designers with artifacts included in the QA-model in their current state. For this approach to an interaction, the developer can use the special set of QA-commands, their sequences and a set of WIQA plug-ins.

The main subset of the positive effects of QA-modeling includes the following:

- controlling and testing the reasoning of the developer with the help of ''collaborative reasoning'' and ''integrated understanding'' included into the QA-models;
- correcting the understanding of designers with the help of comparing it with ''integrated understanding;''
- combining the models of the collective experience with an individual experience for increasing the intellectual potential of the designer on the specific workplace;
- including the individual experience of the developer in accordance with the request on the other workplaces in the corporate network.

As is shown in this scheme, any component of a QA-model is a source of answers that are accessible for the designer as a result of the interactions with the QA-model. At the same time, the potential of the QA-model is not limited by the questions planned when defining and creating the QA-model. Another source of useful effects of QA-modeling is an additional combinatorial ''visual pressure'' of questions and answers, which is caused by the influence on brain processes in their contact with components of the QA-model. No difference depends on who created the QA-model.

There are different forms for building answers with the help of QA-modeling, which are not limited to only linguistic forms. However, the specificity of QA-modeling is defined by the inclusion of additional interactions with ''question-answer objects'' into the dynamics of the integrated consciousness and understanding (into the natural intellectual activity of designers).

The description of any behavioral unit composed of designer interactions with the QA-model in accordance with a specific scenario can fulfill the role of a model of such a designer activity. To distinguish this type of model from other types of models that were used in our approach, they can be named ''QA-models of the designer activity.'' Any such scenario as a specific program reflects designer interactions (actions) aimed at understanding the corresponding task and its solution. In the discussed case, the scenario is a text that comprises instructions that indicate the designer's actions, which should be executed in the reuse of the behavioral unit in the WIQA-medium.

Similar scenarios can be created for human actions that are not limited to the WIQA-medium. Their content, form and appointment are demonstrated by the following technique:

*//Reset of Outlook Express*
*O1. Quit all programs.*
*O2. Start On the menu Run, click.*
*O3. Open In the box regedit, type, and then OK the click.*
*O4. Move to and select the following key:*

*HKEY_CURRENT_USER/Software/Microsoft/Office/9.0/Outlook*

*O5. In the Name list, FirstRunDialog select.*

*O6. If you want to enable only the Welcome to Microsoft Outlook greeting, on the Edit menu, Modify, click the type True in the Value Data box, and then the OK click.*

*O7. If you also want to re-create all of the sample welcome items, then move to and select the following key:*

*HKEY_CURRENT_USER/Software/Microsoft/Office/9.0/Outlook/Setup*

*O8. In the Name list, select and delete the following keys: CreateWelcome First-Run*

*O9. In the Confirm Value Delete dialog box, click Yes for each entry.*

*O.10. On the Registry menu, click Exit.*

*O11. End.*

This technique is chosen to emphasize the following.

1. There are many behavior units that describe human activity in different computerized mediums.
2. Descriptions of similar typical activities help in the reuse of these precedents.
3. Descriptions of techniques have forms of programs (N-programs) that are written in the natural language LN in its algorithmic usage.
4. Such N-programs are made of operators that are fulfilled by humans interacting with the specific computerized system. In the example of the N-program, its operators are marked by the symbol ''O'' with the corresponding digital index.

Thus, there are no obstacles for uploading the N-programs into QA-memory. This method is used for uploading the techniques that support the designer activity in the WIQA-medium.

Thus, the other way of coding the designer activity is connected with its programming in the context of the scientific research on the task. All of the tasks indicated above are uploaded to QA-memory with the rich system of operations with interactive objects of the Z-, Q-, and A-types. Designers have the opportunity to program the interactions with necessary objects. Such programs are similar to the plans of the experimental activity during the conceptual design of the SIS. Operators of programs are placed in Q-objects. Corresponding A-objects are used for registering the facts or features of the executed operations.

Thus, in experimenting with units of their own behavior, the designer has a flexible means for specifying the QA-programs, QA-operators and QA-data that are used in simulating such behavioral units. Experimentation is fulfilled in the form of QA-modeling for solving tasks in conceptual design.

### D. MEANS OF QA-MODELING

Above the QA-model of the task, there is the aspect of QA-reasoning that is registered in QA-memory. However, artifacts of this type are especially useful in cases when they are being created while solving a project task. Therefore, a typical structure of QA-models is specified in the framework form presented in Fig. 5.

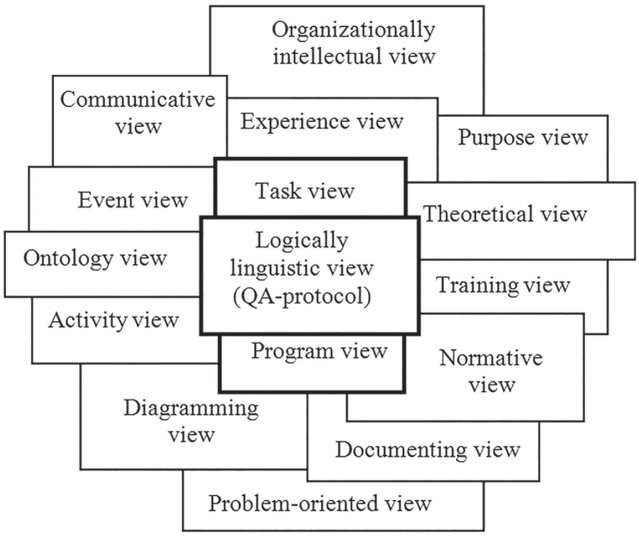


**FIGURE 5.** Framework for QA-models of tasks.

The structure and content of the framework were chosen with an orientation towards the set of typical RUP-tasks, which this technological system gives to designers for their work in the conceptual design stage. Such a decision can be explained in that RUP is the richest source of typical tasks that are confirmed by practice. The conceptual methods of RUP help to build and express the conceptual solution of any task that is involved in the development of SIS.

The framework is destined for building examples of QA-models and adjusting each of them based on the specificity of the corresponding task. At any time during the concept design process, the system of created QA-models should reflect the state of this process from a set of useful viewpoints. For this reason, the framework is specified as the system of views indicated in Fig. 5.

Destinations of viewpoints and the methods for their materialization have been described in our publications [4] and [23]; therefore, in this paper, we shall only emphasize some of the features of the used architecture of the QA-models.

1. The central place in the framework is occupied by the logical linguistic view, which opens a powerful opportunity for registering and using QA-reasoning for different useful purposes. This view specifies QA-memory.
2. Any other view explicitly or implicitly depends on the logically linguistic view, extending its applications in conceptual design.
3. Any view corresponds to the specific operating space for designers, where they solve a set of useful tasks.
4. This work of designers with any view is supported by the corresponding means included in the toolkit WIQA.

The use of mechanisms of views opens the possibility of an interpretation of the WIQA methods as a specialized software intensive system SISQA that is oriented towards workflows ''Interactions with Experience'' in conceptual design. Such understanding has allowed application of the experience of the architectural description for SIS to the toolkit SISQA. The features of WIQA are reflected by its general structure of components, which is presented in Fig. 6.

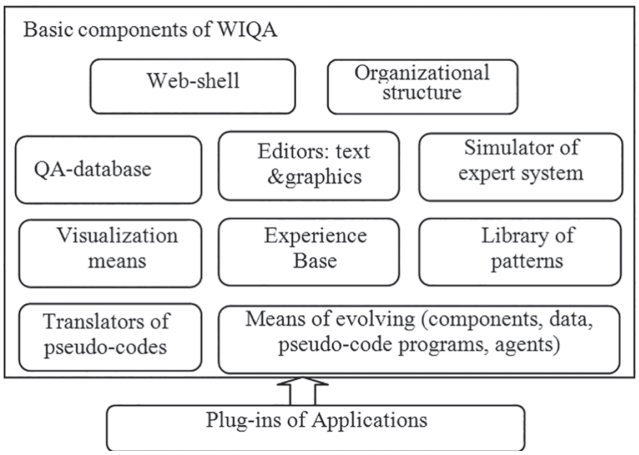


**FIGURE 6.** Generalized structure of WIQA.

The toolkit WIQA is implemented as a client-server system that has been evolved architecturally view by view. The basic programming language is C#. Each view is supported by corresponding plug-ins. By analogy with a Word-processor (for example, Microsoft Word), the toolkit is interpreted as a Question-Answer processor (QA-processor).

## V. PSEUDO-CODE LANGUAGE $L^{WIQA}$

### A. STRUCTURE OF LANGUAGE

QA-reasoning can be used by designers when they create different conceptual models of tasks, for example, in formulating the task statement or in cognitive analysis of the formulated statement or in (pseudo-code) programming the solution plan of the task. The toolkit WIQA supports the creative work of designers with all indicated conceptual modes and conceptual models of the other types.

The specialized pseudo-code language $L^{WIQA}$ has been developed for the use of QA-reasoning in programming the solution plans. This language is oriented towards its use in experiential interactions of designers with accessible experience when they create programs of their own activity and investigate them. Step-by-step, $L^{WIQA}$ has been evolved to a state with the following components.

1. Traditional types of data, such as scalars, lists, records, sets, stacks, queues and the other data types.
2. Data model of the relational type, describing the structure of the database.
3. Basic operators, including traditional pseudo-code operators, for example, Appoint, Input, Output, If-Then-Else, GOTO, Call, Interrupt, Finish and the others operators.
4. SQL-operators in their simplified subset, including Create Database, Create Table, Drop Table, Select, Delete From, Insert Into, and Update.
5. Operators for managing the workflows oriented towards collaborative designing (Seize, Interrupt, Wait, Cancel and Queue).
6. Operators for visualization developed for the creation of the dynamic view of cards presenting QA-units in the direct access of the designer to objects of QA-memory.

The important type of basic operators includes an explicit or implicit command aimed at the execution by the designer of the specific action. Explicit commands are written as imperative sentences in the natural language in its algorithmic usage. When designer interactions with descriptions of questions or answers are used as causes for designer actions, then, such descriptions can be interpreted as implicit commands written in $L^{WIQA}$. For example, textual forms of questions are a very important class of implicit commands.

### B. EMULATION OF DATA AND OPERATORS

In the general case, a QA-program can include data and operators from different enumerated subsets. However, the traditional meaning of such data and operators is only one aspect of their content. The other side is bound with attributes of QA-units in which data and operators are uploaded. As described above, QA-data and QA-operators inherit the attributes of corresponding cells of QA-memory. They inherit not only attributes of QA-units but also their understanding as ''questions'' and ''answers.''

Originally, QA-data had been suggested and developed for real-time work with such interactive objects as ''tasks,'' ''questions'' and ''answers,'' which were stored in the QA-database and used by designers in the corporate network. It is necessary to recall that ''task'' is a type of question and ''decision of the task'' is an answer to a question.

On the logical level, any QA-data can be interpreted as the specialized hierarchical model of data emulated by means of the relational model of the data. Two hierarchical trees of data, the units of which are connected as questions and answers, is one of the specificities of QA-data. The general version of QA-data includes the dynamic tasks tree of the units, which are united with a system of QA-models for corresponding tasks.

Let us recall that any unit of such a model is the interactive object with a unique name and symbolic expression, which are visually accessible to designers in the task tree or in the corresponding QA-protocol. Other characteristics (for example, such basic attributes as the name of the creator, time attributes, indicator of changes, and attribute of inheritances) have been discovered and used in different planned actions with the data unit.

One way of broadening the interpretation of QA-data is connected with the abstract type of data with named attributes and features, including the accessible set of commands.

Such an interpretation allows for the developers to use the abstract QA-type for the emulation of types of data that are necessary for pseudo-code programming.

The emulation is based on the use of Q-objects and corresponding A-objects for the following presentation of any normative types:

- "question" – "name of the variable for the simple type of data," and "answer" – "its value;"
- "definite composition of questions" – "typical data" (for example, array, record, set, array of records or table, stack, queue and others types of composite data) and "corresponding composition of answers" – "its value."

There are sufficient reasons for the interpretation of variables (names of variables) as questions and their values as corresponding answers. As a result, QA-data can be used for emulating data of many known types. As told above in WIQA, there is a special mechanism for assigning the necessary characteristics to a specific unit of QA-data. It is the mechanism of additional attributes (AA) that provides the possibility of expanding the set of basic attributes for any Z-, Q- or A-object stored in the QA-database.

The mechanism of AA implements the function of the object-relational mapping of QA-data to program objects with planned characteristics. One version of such objects is classes in C#. The other version is fitted for pseudo-code programming. The scheme that is used in WIQA for the object-relational mapping is presented in Fig. 7.

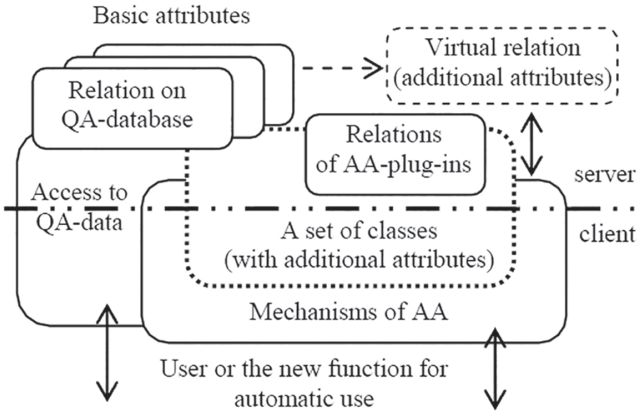


**FIGURE 7.** Creation of additional attributes.

The use of the AA is supported by the specialized plug-ins embedded in WIQA. This plug-in helps the designer declare the necessary attribute or a group of attributes for specific QA-units. At any time, the designer can view declared attributes for the chosen unit. Any necessary actions that have assigned additional attributes can be programmed in C# or in the pseudo-code language $L^{WIQA}$.

Broadening of the abstract type of QA-data by means of additional attributes helps to emulate any traditional data types, such as scalars, arrays, records lists, and others. Moreover, means of additional attributes open the possibilities for assigning to simulated data their semantic features. An example of specifying the array with elements of the integer type is presented in Fig. 8, where a set of additional attributes is used for translating the array declarations to computer code.

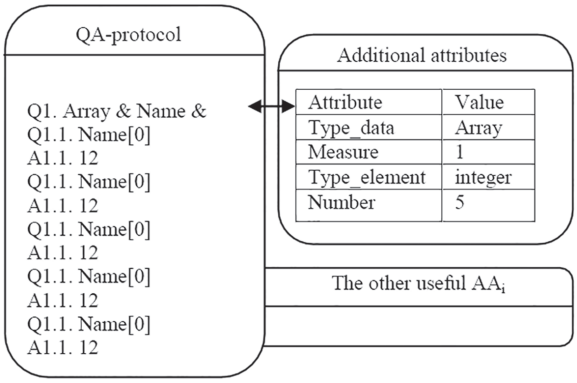


**FIGURE 8.** Declaration of array.

Attributes that are assigned for the array are visually accessible for the designer at any time and can be used for more than for translating. The designer can add useful attributes to the set of array attributes, for example, to describe its semantic features, which will be checked in creating and executing the QA-program. Any designer can create any necessary type of data, for C#-programming as well as for pseudo-code programming.

### C. EMULATION OF OPERATORS

The second type of pseudo-code strings is intended for writing the commands (operators). Similar to for QA-data, we can define the next interpretations for operators:

- "question" is "a symbolic presentation of an operator;"
- "answer" is connected with the results of the operator execution.

In other words, the symbol string of the "question" can be used for writing (in this place) the operator in the pseudo-code form. The fact or the result of the operator execution will be marked or registered in the symbol string of the corresponding "answer."

The following remarks explain the specificity of QA-operators and their use.

1. Any sentence in any natural language includes the interrogative component, which can be indicated explicitly or implicitly. In QA-reasoning, this component is used obviously, while in the pseudo-code operator, the question is presented implicitly.
2. Named interpretation opens the possibility of registering pseudo-code programs in QA-memory in the form of programmed QA-models of the corresponding tasks.
3. In this case, any pseudo-code operator presented by the pair of coordinated interactive objects of Q- and

A-types is written on the ''surface'' of the corresponding QA-unit in QA-memory.

4. Thus, the used QA-unit can be interpreted as the ''material for writing'' of the corresponding operator of the source pseudo-code. This material has useful properties, which are presented figuratively by attributes of the QA-unit in Fig. 9.

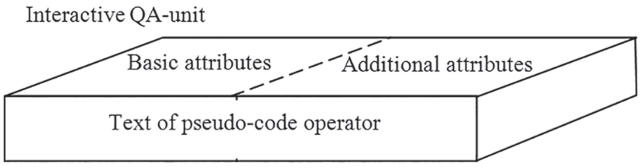


**FIGURE 9.** Writing of the pseudo-code operator.

This ''material'' comprises visualized forms for writing the string symbols that were originally intended for registering the texts in the field ''textual description'' of the corresponding QA-unit. The initial applicability and features of such a type of strings are inherited by data and operators of pseudo-code programs. It is possible to assume that data and operators are written on ''punch-cards,'' the features of which (basic and useful additional attributes of corresponding QA-units) can be accessible for their processing together with textual descriptions if it is necessary.

Thus, the traditional grammar of pseudo-code languages can be extended by the use of indicated attributes of QA-units and the operations with them. This reason was one of the principal reasons for qualifying QA-programs as the new type of pseudo-code programs.

### D. EXAMPLES OF QA-PROGRAMS

QA-programs are implemented for the programming of the condition part and the reaction part of any precedent separately. The condition part should help to solve the task of the precedent choice from the base of precedents. The typical scheme of QA-programs for the conditioned parts has the following view:

*QA-PROGRAM_1 (condition for the access to the precedent):*
*N1 Variable &V_1& / Comment_1: symbolic indicator "N"(Name) is used as a type of "question"*
*V1 Value of V_1. / "V" indicates value of variable*
*N2 Variable &V_2& / Comment_2?*
*V2 Value of V_2.*
.....
*NM Variable &V_M& / Comment_M?*
*VM Value of V_M.*
*ON F =Logical expression (&V_1&, &V_2&,...,&V_M&)*
*EN Value of Expression.*
*OP Finish.*

It is necessary to notice that the designer can build or modify or fulfill (step by step) a specific example of similar programs in real-time work with the corresponding precedent, which the designer creates or reuses. In the presented typical scheme, the logical expression of the choice is programmed as the program function (QA-function).

The reaction part of the precedent very often has a technique type that is a similar technique for Reset of Outlook Express described above. In other cases, the designer can program the reaction part in the prototype form.

The next fragment of a QA-program is used in plug-ins called ''System of Interruptions'' for the calculation of a priority of the interrupted QA-programs being executed by the designer in parallel:

*O 1.11 Procedure &DiscardPriority&*
*O 1.11.1 &P& := &Pmax&/*
*O 1.11.2 Label &DP1&/ "O 1.11.2"-unique index name (Address)*
*O 1.11.3 &Priority& := &P&*
*O 1.11.4 CALL &GetTaskByPr&*
*O 1.11.5 &base& − > &TaskPriority& := &base& − > &TaskPriority&+1*
*O 1.11.6 CALL &ChangeTask&*
*O 1.11.7 &P& := &P& - 1*
*O 1.11.8 IF &P&< &base& −> &Pmin& THEN &base& −> &NewPriority& := &Pmin& ELSE GOTO &DP1&*
*O 1.11.9 ENDPROC &DiscardPriority&.*

This QA-procedure is translated by the compiler (not by the interpreter) because it is processed by the computer processor (not by the I-processor). Therefore, A-lines of operators are excluded from the source code.

It is necessary to mark that a creation and execution of any QA-program is implemented in an instrumental environment, which includes two translators (interpreter and compiler), editor, debugger and a number of specialized utilities for working with data declarations [2].

### E. INSTRUMENTAL ENVIRONMENT OF QA-PROGRAMMING

The language $L^{WIQA}$, as in any other applied language, is not separated from the means of its usage. Methods of QA-programming have been developed and embedded into WIQA during its evolution. The creation and use of QA-programs are fulfilled in the instrumental environment presented in Fig. 10.

Creation of a specific QA-program involves beginning from the choice of the point in the tasks tree and the declaration of the new task for this program. The index name of this task (1) will be used as the initial address for computing the index names for any line of source code of a QA-program that is written in the area (2) of the text editor. The indexed copy of the source code is registered in the editor memory, and it is visualized in the area (3). After saving the current state of the source code, its indexed copy transfers (4) to the QA-database (5). At any time, any QA-program from the database can be uploaded to the editor.

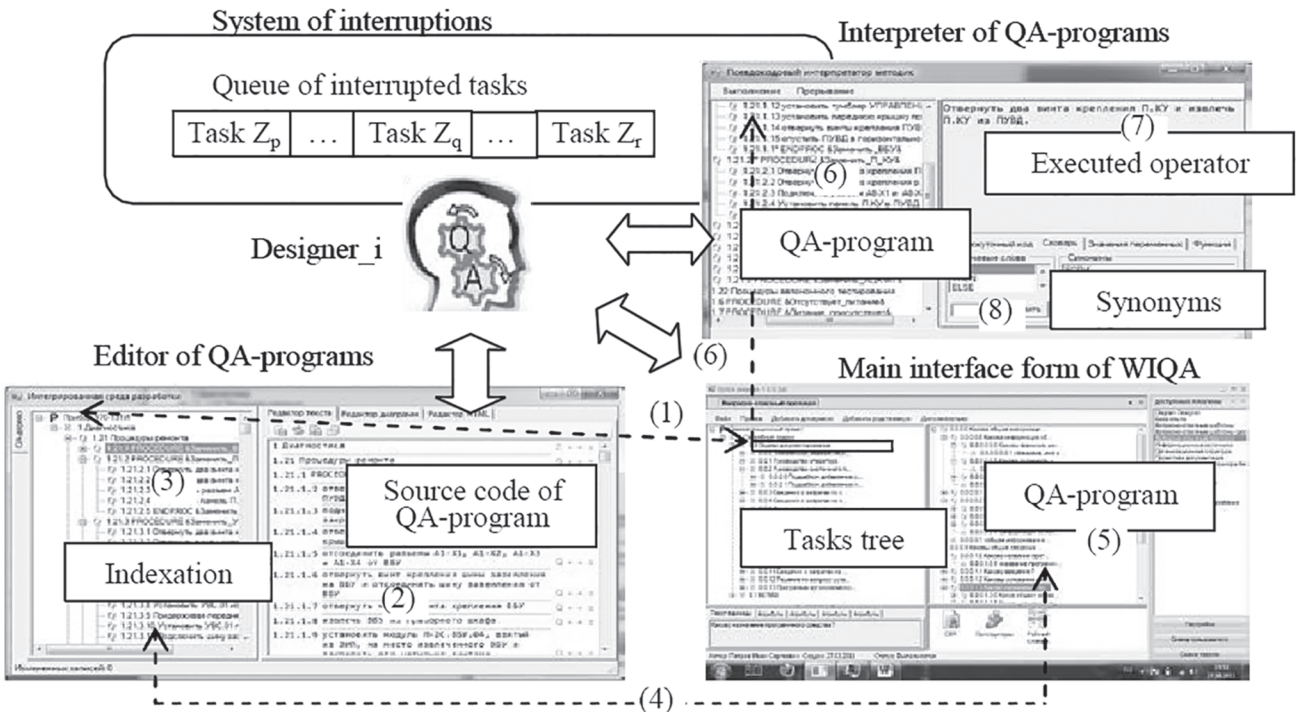


**FIGURE 10.** Environment of QA-programming.

Any QA-program in any of its states can be uploaded (6) to the interpreter for execution. Any executed operator of a QA-program is visualized in the special area (7) of the interpreter, and at any time, the designer can declare a new synonym for the chosen keyword (8) or variable. There are other useful possibilities that are accessible for the designer in the main interface forms presented above in Fig. 10.

The reality of the designer activity is that there is parallel work involving many simultaneous tasks. Therefore, a special system of interruptions is included in WIQA. This system provides the opportunity to interrupt any executed task or QA-program (if it is necessary), to work with other tasks or QA-programs. The interruption subsystem supports the return to any interrupted task or QA-program to the point of the interruption.

## VI. SIMULATING THE DESIGNER'S BEHAVIOR

### A. PREPARATION OF EXPERIMENTS

The principal feature of the proposed approach is an experimental investigation by the designer into the programmed behavior, which has led to the conceptual solution of the appointed task. Any solution of such a type should demonstrate that its reuse meets the necessary requirements when any designer of the team will act in accordance with QA-program of the investigated behavior.

As described above, to achieve the goals, the designer should work in a way similar to a scientist who prepares and conducts experiments with the behavior units of the M- or P-types. In the discussed case, the designer will experiment in the environment of the toolkit WIQA. In this environment, to prove that the aim of an experiment has been achieved, the designer has the possibility of experimenting with any QA-operator of an investigated QA-program and/or with any group of such QA-operators or with the QA-program as a whole. Describing the experiment for reuse, the designer should register it in an understandable form for the other members of the team.

To begin a specific experiment, the initial text of the QA-program should be built. In the general case, such a project would include the following steps.

1. Formulation of the initial statement of the task.
2. Cognitive analysis of the initial statement with the use of QA-reasoning and registering it in QA-memory.
3. Logical description of the "cause-effect relation" reflected in the task.
4. Diagrammatic presentation of the analysis results (if it is necessary or useful).
5. Creation of the initial version of the QA-program.

The indicated steps are fulfilled by the designer with the use of the accessible experience, including the personal experience and useful units from the Experience Base of WIQA.

### B. EXPERIMENTING WITH THE QA-PROGRAM

Only afterward can the designer conduct the experiment, interacting with the QA-program in the context of the accessible experience. The specificity of interactions can be clarified on examples of QA-operators of any QA-program or its fragment, for example, the following fragment of QA-program coding the well-known method of SWOT-analysis (Strengths, Weaknesses, Opportunities, and Threats):

*Q 2.5 PROCEDURE &SWOT main&*
*Q 2.5.1 &t_str& := QA_GetQAText(&history_branch_qaid&)*
*Q 2.5.2 SETHISTORYENTRIES(&t_str&)*
*Q 2.5.3 CALL &ShowHistory&*
*Q 2.5.4 IF &LastHistoryFormResult& == −1 THEN RETURN*
*Q 2.5.5 IF &LastHistoryFormResult& == 0 THEN &current_action_qaid& := QA_CreateNode(&current_project&, &history_branch_qaid&, 3, "") ELSE &current_ action_ qaid& := &LastHistoryFormResult&*
*Q 2.5.6 &t_str& := QA_GetQAText(&current_action_qaid&)*
*Q 2.5.7 SWOT_DESERIALIZE(&t_str&)*
*Q 2.5.8 &t_int& := SWOT_SHOWMAINFORM()*
……..
*Q 2.5.14 FINISH*

This source code demonstrates an often-used syntax, but features of the code are opened in interactions of the designer with it. Conditions and methods of experimenting are shown in Fig. 11, where one of the operators (with address name Q2.5.2) is shown in the context of previous and subsequent operators.

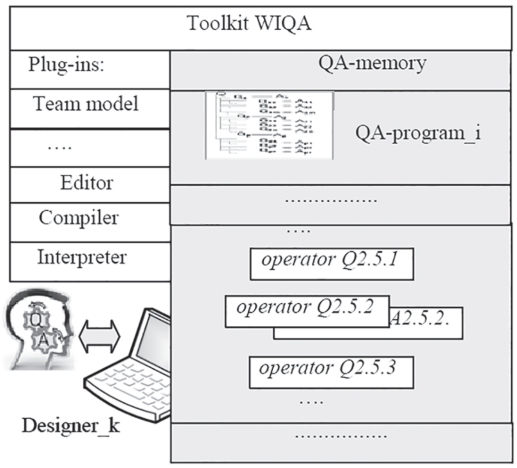


**FIGURE 11.** Experimenting of designer with QA-program.

Any QA-program is executed by the designer step-by-step, in which each step is aimed at the corresponding QA-operator. In this study, the designer uses the plug-in "Interpreter" embedded into the toolkit.

Interpreting the current operator (for example, Q2.5.2), the designer can fulfill any actions until its activation (for example, to test existing circumstances) and after its execution (for example, to estimate the results of the investigation), using any means in the toolkit WIQA. When the designer decides to start this work with the QA-operator, this work can include different interactive actions with it as with corresponding QA-units or with their elements. The designer can analyze values of their attributes and make useful decisions.

Moreover, the designer can appoint the necessary attributes for any QA-operator and for any unit of QA-data at any time. In accordance with appointments, the designer can include changes in the source code of the QA-program being executed (investigated). Such work can be fulfilled as in QA-memory, with the help of the plug-ins "Editor."

The current QA-program or its fragments can be executed or used step-by-step by the designer or automatically as a whole with the help of the plug-in "Compiler." Therefore, all of this work described above with the QA-operator can be used for any of the groups and for any QA-program as a whole. For this reason, the execution of QA-operator by the designer is similarly experimentation. Thus, the designer has a flexible possibility to perform experimental research on any task that is solved conceptually. This feature is the principal feature that distinguishes pseudo-code QA-programs from programs written in pseudo-code languages of different types, including the class of Domain-Specific Languages [24].

The specificity of the described type of designer activity is this work controlled by the QA-program and executed by the designer interacting with the accessible experience. To underline this specificity, the specialized role of "intellectual processor" was constructively defined and is effectively supported in the use of WIQA [9]. This role is added to the other types of roles that applied in the concept design [8].

### C. DESCRIPTION OF EXPERIMENTS

As described above, any experiment that is conducted should be presented by the designer in an understandable and reusable form. In the offered version of experimentation, the function with such a form is fulfilled by the typical integrated model of the precedent, which is shown in Fig. 12.

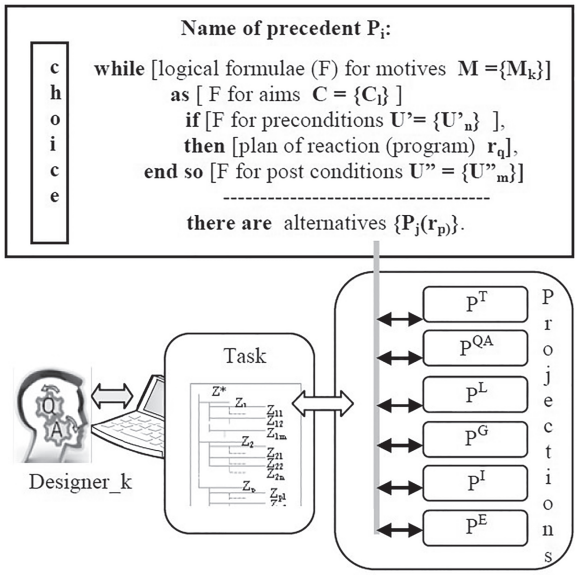


**FIGURE 12.** Framework of precedent model.

The scheme, which satisfies the function of framework F(P) for models of precedents, allows integrating the very useful information that accompanies the experiment process in its actions, as indicated above.

The central position in this model is occupied by the logical scheme of the precedent. The scheme explicitly formulates the ''cause-effect regularity'' of the simulated behavior of the designer. Framework F(P) includes the following components:

- textual model $P^T$ of the solved task;
- its model $^{PQA}$ in the form of registered QA-reasoning;
- logical formulae $P^L$ of the modeled regularity;
- graphical (diagram) representation $P^G$ of the precedent;
- pseudo-code model $P^I$ in QA-program form;
- the executable code $P^E$.

Any component or any of their group can be interpreted as projections of F(P), the use of which allow us to build the precedent model in accordance with the precedent specificity. However, in any case, the precedent model should be understandable to its users.

All of the built models of precedents are divided into two classes, one of which includes models that are embedded in the Experience base of WIQA, which is used by the team, not just in the current project. The second class includes models that are only for the current project.

## VII. INTELLECTUAL PROCESSOR

Any designer playing the role of intellectual processor uses question-answer reasoning and its models for access to information on the prior experience.

The essence of the I-processor is defined by the features enumerated below.

1. First, the I-processor is a role that is played by any designer solving the appointed tasks in the conceptual designing of the SIS. In the life cycle of any SIS, the conceptual stage is an area of intensive modeling of the tasks being solved.
2. I-processors are intended for experimenting with tasks for which finding the solution is problematic without the explicit real-time access to the personal experience and/or collective experience and/or models of useful experience.
3. During its activity, an I-processor interprets tasks as precedents and interacts with experience units similar to with models of precedents.
4. The real time work of any I-processor is accompanied by QA-reasoning, and its models are registered in QA-memory of the toolkit WIQA.
5. QA-reasoning is used by I-processors for creating QA-models of tasks in different forms, including their versions as QA-programs.
6. The use of QA-reasoning in interactions of I-processors with QA-programs is implemented in the pseudo-code language $L^{WIQA}$. The knowledge and effective use of this language by I-processor is a very important feature.
7. In a general case, the activity of I-processor is similar to the experimental activity of the designer who creates the QA-model of the task for experimenting with the prototype of its solution. In such experimental work with the tasks, I-processor can use QA-modeling and QA-programming.

As stated above, the use of QA-programs by I-processor for experimenting with tasks is the most essential of its features. For this reason, we created the library of the specialized QA-programs, which provides some versions of such experimenting. This library includes a number of QA-techniques for cognitive task analysis, decision-making and typical procedures of estimation.

In WIQA, fulfilling the role of ''I-processor'' is automated with the use of computer methods (K-processor) and a set of methods that includes rules of behavior and specialized programs. From this point of view, the WIQA is a specialized processor of the question answer type (QA-processor) that supports experiential interactions between I-processors and K-processors during the collaborative designing of SISs. Combining the named processors is schematically presented in Fig. 13.

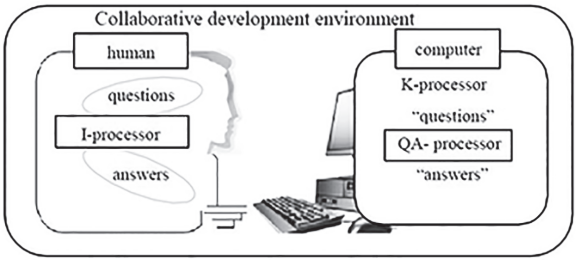


**FIGURE 13.** Interactions of I-processor with QA-processor.

The scheme of the processors allows separating this work with QA-reasoning in the described versions of their use from the designer's activity in playing the other roles supported by the K-processor.

It is necessary to note that our materialization of I-processor implements by means of WIQA, but it is our solution. In principle, the requirements to I-processor can be implemented on the basis of instrumental methods that are distinct from the toolkit WIQA.

## VIII. COLLABORATIVE ACTIVITY OF DESIGNERS

Conceptual designing of SIS is a collaborative activity of designers working in coordination. Typical units of such an activity are the workflows, any of which can be simulated with the empirical approach. In this case, experiments will be fulfilled collaboratively by the group of designers, while playing the roles of I-processors. For QA-programming of workflows, the specialized subset of operators has been included in the language $L^{WIQA}$. The generalized scheme of the collaborative actions of the designers is presented in Fig. 14.

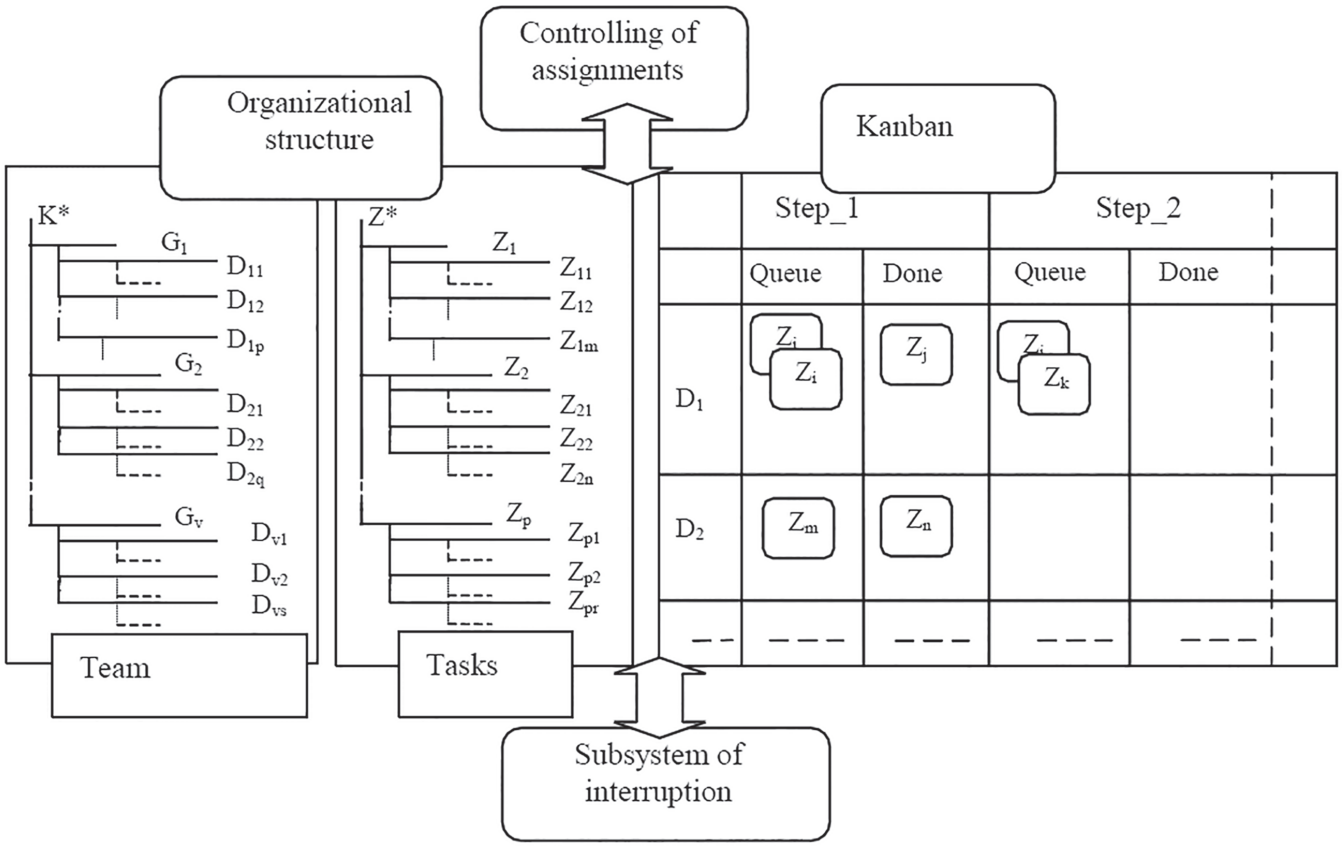


**FIGURE 14.** Scheme of collaborative activity of designers.

In this scheme, the subsystem "Organizational structure" supports the real-time assigning of tasks to members of the designer team. The copy of the team model (K*, $\{G_v\}$, $\{D_{vs}\}$), which specifies groups $\{G_v\}$ of designers $D_{vs}\}$, can be uploaded in QA-memory in the form presented in Fig. 3.

The group manager uses the subsystem "Controlling of assignments" for binding any assignment with the planned time of its fulfillment by the designer who is responsible for the assigned task. Subsystem "Kanban" [25] automatically reflects the steps of the workflow execution with the help of visualizing the current state of the queues of the tasks; this approach helps to control the process of designing.

Pseudo-code programming of workflows is based on using the library of workflow patterns [22]. Coding the units of this library illustrates the example of the workflow pattern "Simple Merge." This pattern is described by the statement: The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch).

For the three tasks Z1, Z2, and Z3, this pattern has the following view:

*Q.3.5 PROCEDURE &Simple_Merge&// index name are inherited also but from library of pattern*
*Q 3.5.1SET &out&, 4; &ins[0]&, 1; &ins[1]&, 2; & outgroup[0]&, 1; &outgroup[1]&, 2; &cnt&, 0*
*Q 3.5.2 LABEL &L1&*
*Q 3.5.3 SEIZE &outs[&cnt&]&, &outgroup[&cnt&]&*
*Q 3.5.4 INC &cnt&*
*Q 3.5.5 TEST L, &cnt&, &ins&.length &L1&*
*Q 3.5.6 LABEL &L2_1&*
*Q 3.5.7 SET &cnt&, 0;*
*Q 3.5.8 LABEL &L2&*
*Q 3.5.9 TEST E, &ins[&cnt&]&, &ins[&cnt&]&.state, DONE &L3&*
*Q 3.5.10 INC &cnt&*
*Q 3.5.11 TEST L, &cnt&, &ins&.length &L2_1&*
*Q 3.5.12 TRANSFER &L2&*
*Q 3.5.13 LABEL &L3&*
*Q 3.5.14 QUEUE &out&, TRUE*
*Q 3.5.15 SET &ins[&cnt&]&.state, WAITING*
*Q 3.5.16 TRANSFER &L2_1&*
*Q 3.5.17 ENDPROC &Edit_Assignment&*

In the discussed case, the queues with which the designer works is expediently interpreted as a special type of program (M-programs), which manages the activity of the designer. Any unit of the queue in the corresponding M-program includes the names of the corresponding task and attributes that specifying the conditions in which this work with this task can be begun or interrupted. Any unit of such a queue is

interpreted as an operator of an M-program. The interaction of the I-processor with M-programs is figuratively presented in Fig. 15.

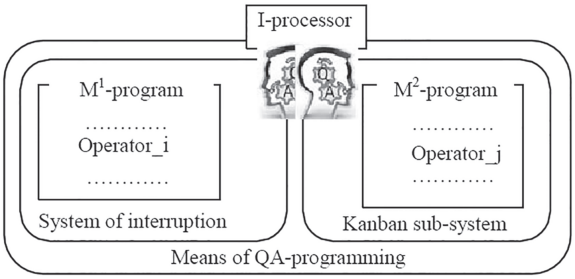


**FIGURE 15. Interactions of I-processor with M-programs.**

There is a possibility of using two types of M-programs in the WIQA-environment. The first type $M^1$ provides the pseudo-parallel solving of tasks (QA-programs) by I-processor when any of the tasks can be interrupted by I-processor or by external reasons in the process of obtaining its solution. Such an opportunity is supported by the plug-in "System of interruption."

Any $M^2$-program manages the execution of tasks by the specific I-processor in the workflows, which are processed collectively with the use of other means presented in Fig. 14.

## IX. CONCLUSION

The approach described in this paper suggests a system of methods that simplify the complexity of designers' interactions with project tasks in the conceptual designing of an SIS. The simplification is caused by using, in the designers' activity, useful analogies with this work of scientists when conducting experiments. Emulating scientists is accomplished by designers while investigating their own behavior in the processes of obtaining task solutions. Moreover, they simulate such behavior with the help of pseudo-code programs that describe the plans for experimentation. Thus, designers investigate the programmed plans of the experiments that they prepare, and they conduct and describe the results in understandable and checkable forms, for later reuse.

In the experiments, the investigated behavioral units are modeled as precedents. Such a form for a human activity is natural because the intellectual processing of precedents comprises the base of the human experience. In experimentation, the designers evolve accessible experience by using real-time interactions with the current state. This feature has found normative specifications in the role of "intellectual processor," which is played by designers and is being supported by the toolkit WIQA. In the collaborative way-of-working, this aspect can be used in addition to any other aspect of the technology that is applied in the conceptual design process.

This toolkit opens the possibility for the separate execution of any operator by the designer playing the role of the intellectual processor. Before and after the execution of any operator of any QA-program, the designer can check or investigate its preconditions and post-conditions. Moreover, the investigated operator can be changed and evolved syntactically as well as semantically, for example, with the help of additional attributes.

Debugged QA-programs are the source of resources of the M- and P-types, which promotes the simplification of the complexity that is involved in their reuse. The possibility of experimenting is supported by the special library of QA-programs destined for cognitive task analysis, problem-solving and decision-making included in the named toolkit. Suggested means are used in one project organization in creating a family of SISs.

## REFERENCES

[1] K. E. Emam and A. G. Koru, "A replicated survey of it software project failures," *IEEE Softw.*, vol. 25, no. 5, pp. 84–90, Sep. 2008.

[2] I. Jacobson, P.-W. Ng, P. McMahon, I. Spence, and S. Lidman, "The essence of software engineering: The SEMAT kernel," *ACM Queue*, vol. 10, no. 10, pp. 1–12, Oct. 2012.

[3] M. Li and P. Vitanui, *An Introduction to Kolmogorov Complexity and Its Applications*, 3rd ed. New York, NY, USA: Springer-Verlag, 2008.

[4] P. Sosnin, "Conceptual solution of the tasks in designing the software intensive systems," in *Proc. MELECON*, May 2008, pp. 293–298.

[5] J. M. Pattit and D. Wilemon, "Creating high-performing software development teams," *R&D Manag.*, vol. 35, no. 4, pp. 375–393, Sep. 2005.

[6] Y. Cho, "The state of the art of action learning research," *Adv. Develop. Human Resour.*, vol. 12, no. 2, pp. 163–180, Apr. 2010.

[7] *Managing for the Sustained Success of an Organization—A Quality Management Approach*, Standard ISO 9004-2009, 2009.

[8] P. Borges, R. J. Machado, and P. Ribeiro, "Mapping RUP roles to small software development teams," in *Proc. ICSSP*, Jun. 2012, pp. 190–199.

[9] P. Sosnin, "Experiential human-computer interaction in collaborative designing of software intensive systems," in *Proc. 11th Int. Conf. Softw. Methodol. Tech.*, Oct. 2012, pp. 180–197.

[10] F. Karray, M. Alemzadeh, J. A. Saleh, and M. N. Arab, "Human-computer interaction: Overview on state of the art," *Smart Sens. Intell. Syst.*, vol. 1, no. 1, pp. 138–159, Jun. 2008.

[11] I. Jacobson, P.-W. Ng, and I. Spence, "Enough of processes-let's do practices," *J. Object Technol.*, vol. 6, no. 6, pp. 41–67, Jul./Aug. 2007.

[12] C. Cares, X. Franch, and E. Mayol, "Perspectives about paradigms in software engineering," in *Proc. 2nd Int. Workshop Philisophiocal Found. Inf. Syst. Eng.*, 2006, pp. 737–744.

[13] D. R. Jeffery and L. Scott, "Has twenty-five years of empirical software engineering made a difference," in *Proc. 2nd Asia-Pacific Softw. Eng. Conf.*, 2002, pp. 539–549.

[14] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen, "The future of empirical methods in software engineering research," in *Proc. IEEE Workshop Future Softw. Eng.*, May 2007, pp. 358–378.

[15] V. R. Basili, M. Lindvall, and P. Costa, "Implementing the experience factory concepts as a set of experience bases," in *Proc. SEKE*, 2001, pp. 102–109.

[16] P. H. Southekal and G. Levin, "Formulation and empirical validation of a gqm based measurement framework," in *Proc. 11th Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2011, pp. 404–413.

[17] S. Xu and V. Rajlich, "Dialog-based protocol: An empirical research method for cognitive activity in software engineering," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, Nov. 2005, pp. 397–406.

[18] B. Webber and N. Webb, "Question answering," in *Handbook of Computational Linguistics and Natural Language Processing*. Oxford, U.K.: Blackwells, Jun. 2010, pp. 630–655.

[19] S. Henninger, "Tool support for experience-based software development methodologies," *Adv. Comput.*, vol. 59, no. 1, pp. 29–82, Jan. 2003.

[20] (2013). *IBM Rational Unified Process (RUP)* [Online]. Available: http://www-01.ibm.com/software/rational/rup/

[21] M. Held and W. Blochinger, ''Structured collaborative workflow design,'' *Future Generat. Comput. Syst.*, vol. 25 no. 6, pp. 638–653, Jun. 2009.

[22] W. M. P. Van der Aalst and A. H. M. Hofstede, ''Workflow patterns put into context,'' *Softw. Syst. Model*, vol. 11, no. 3, pp. 319–323, Jul. 2012.

[23] P. Sosnin, ''Pseudo-code Programming of designer activity in development of software intensive systems,'' in *Proc. 25th Int. Conf. Ind. Eng. Other Appl. Appl. Intell. Syst.*, Jun. 2012, pp. 457–466.

[24] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel, ''Design guidelines for domain specific language,'' in *Proc. 9th OOPSLA Workshop Domain-Specific Model.*, Oct. 2009, pp. 7–13.

[25] J. X. Wang, ''Kanban: Align manufacturing flow with demand pull,'' in *Chapter in the Book, Lean Manufacturing Business Bottom-Line Based*. Cleveland, OH, USA: CRC Press, 2010, pp. 185–204.

**PETR SOSNIN** (M'04) was born in Ulyanovsk, Russia, in 1945. He graduated from Ulyanovsk State Technical University, Ulyanovsk, in 1968, and the Doctoral degree from the Moscow Aviation University, Moscow, Russia, in 2004. He is the Head of the Computer Department, Ulyanovsk State Technical University, since 1990. He is a member of ACM and the Russian Association of Artificial Intelligence.

He has more than 30 years of experience in the domain of artificial intelligence in its applications to the computer aided design of software intensive systems. He is the author of 11 books and more than 250 scientific publications in Russian and English languages. He was a Scientific Advisor in 20 Ph.D.-dissertations. His current research interests include collaborative activity of designers in computerized technological environments, modeling of professional experience, and human-computer interactions in problem-solving. He is the Honored Specialist of High Education in Russian Federation.