UDC 004.415.2

TETE-A-TETE PROJECT:
SOFTWARE ENGINEERING TOOLS SUPPORTING UNDERSTANDING

[1] *V.P. Hrytsay*, [2] *L.M. Zakhariya*

[1]National University, Sacramento, California, USA,vhrytsay@gmail.com,

[2]National University "Lvivska Polytechnika",Lviv, Ukraine,zlm.lviv@gmail.com

Software Understanding is necessary and the most important condition of essential reducing of software development cost. In this paper Tete-A-Tete Project is described which offers a radical rethinking of Software Development process and focuses on creating Understanding-Oriented Automated Software Engineering Tools. The project is based algebraic approach to the presentation of programs, which opens up the possibility of transformation and optimization programs. Implementation of the Tete-A-Tete Project will make it possible to build high quality software more quickly and with lower cost, than previously possible; will have essential impact on Software Reengineering and will lead to rethinking and reinvention of the Chief Programmer Team in the form of the Software Director Team.

Познаваемость методов и средств разработки программного обеспечения является необходимым и наиболее важным условием существенного снижения стоимости программных систем. В этой работе описывается проект Tete-A-Tete, который предлагает радикальное переосмысление процесса разработки программного обеспечения и направлен на создание ориентированных на познаваемость автоматизированных средств разработки программных систем. В основе проекта лежит алгебраический подход к представлению программ, который открывает возможности трансформации и оптимизации программ. Реализация проекта Tete-A-Tete позволит создавать программное обеспечение более высокого качества , быстрее и с меньшими затратами, чем это было возможно ранее; окажет влияние на Реинжининг Программного Обеспечения и приведет к переосмыслению и возврату к использованию Бригады Главного Программиста в виде Бригады Программного Режисера.

Introduction

"Programming is understanding"
Kristen Nygaard

The problem of essential reducing software cost (Software Cost Problem) is very difficult unsolved problem. The most software lifecycle costs occur after initial development and deployment. Software systems permanently evolve and their transformations are required to make them adaptable to changing requirements and environments.

Software Engineering is and will always be an inherently human activity. This is why human understanding of software systems and processes of their creation and transformation - Software Understanding – is an extremely important aspect of Software Engineering assisting to overcome the difficult and complexity of large scale software systems.

We assert that Software Understanding is necessary and the most important condition of solving Software Cost Problem.

In this paper we describe some results obtained during work of many years on Tete-A-Tete Project (Tete-A-Tete).

Tete-A-Tete focuses on creating Understanding-Oriented Automated Software Engineering Tools implementing the idea of Michael Jackson that "…enabling and supporting human understanding must be the goal of the software engineering tools" [1].

Tete-A-Tete offers a radical rethinking of Software Development:

- Software Design should be completely separated from Software Implementation (Software Construction).
- Software Director (Software Régisseur), a new stakeholder of Software Engineering, is introduced, unifying Software Development under the vision of a single person.
- Software Director, equipped with a universal Software Design Description Language (Tete-A-Tete_Language), should be solely responsible for Software Design and the Conceptual Integrity of the Software System [2].
- Software Director should also be equipped with Software Development Tools (Tete-A-Tete_Designer) supporting Tete-A-Tete_Language and enabling Automated Generation of up to 60-75% lines of Software Code as well as Automated Generation of Database Schemas and User Acceptance Tests for various Implementation Environments.

Implementation of Tete-A-Tete Project should:

- Make it possible to build software more quickly, with lower cost and higher assurance of quality, than previously possible.
- Have broader impact on Software Reengineering making it much easier for software systems developed in Tete-A-Tete_Designer;
- Lead to reinvention of the Chief Programmer Team Management approach [3, 4].

Prehistory of the Tete-A-Tete Project

Tete-A-Tete focuses on creating Understanding-Oriented Automated Software Development Tools. It is a logical evolution of the ideas and accomplishments, developed in the successful MULTIPROCESSIST Project (Glushkov Institute of Cybernetics NAS Ukraine, 1978-1988) [5-8].

Dr. Tseitlin G.E. was the initiator and leader of R & D team, working on this Project, which:

- studied Problem of weak Program Understanding;
- created Requirements for Program Design Languages;
- created SAA/1 Algorithms Design Language, based on Glushkov-Tseitlin Algebra of Algorithms;
- developed PL/I-program synthesizer with SAA/1 as a source language;
- (Later synthesizers for PASCAL, FORTRAN, COBOL, and ASSEMBLER (for IBM and PDP-11 platforms) have been developed);
- installed PL/I-program synthesizer in many companies and organizations.

MULTIPROCESSIST system proved its efficiency when developing programs with complicated logic and relatively simple data structures. Program Understanding was essentially improved (thanks to SAA/1) and thereby Program Maintenance became more qualitative and rapid [5].

On average, for every synthesized program 66% lines of code were synthesized automatically.

In brief, synthesis of a PROGRAM in some programming language L in the MULTIPROCESSIST system looks as follows.

Consider the following algorithmic description in SAA/1 Language (called SAA-schema):

```
    schema MMM = “Algorithm of solution of the  Ax^2+Bx+C=0 equation”;
“Analysis of the equation Ax^2+Bx+C=0 “   is-defined-as
        if ‘Coefficient A <> 0’
        then “Solution of the quadratic equation Ax^2+Bx+C=0”
        else “Solution of the linear equation Bx+C=0”;
“Solution of the quadratic equation Ax^2+Bx+C=0” is-defined-as
        “Calculate: Z=-B/2A”
        *
        “Calculate Discriminant: D= Z^2 – C/A”
        *
        if ‘Discriminant D > 0’
        then “Calculate 2 different real roots”
        *
        if ‘Discriminant D = 0’
        then   “Calculate 2 equal real roots”
        *
        if    ‘Discriminant D < 0’
        then “Calculate 2 different complex roots”;
    “Solution of the linear equation  Bx+C=0” is-defined-as
        if  ‘Coefficient B <> 0’
        then “Single Root: X = - C/B”
        else
                if ‘Coefficient C <> 0’
                then “Equation has no roots”
                else “ Equation has infinitely many roots”
    end schema   MMM
```

(Asterisk ‘*’ denotes sequence operation).

At first glance this text looks like pseudocode, but in reality this is the description in SAA/1 Language. This Language is based on Glushkov-Tseitlin Algebra of Algorithms - Systems of Algorithmic Algebras (SAA) - which is a universal algorithmic formal system, equivalent to Turing Machines, Recursive Functions, Markov Algorithms etc., and conceptually adequate to Structured Programming.

SAA/1 Language has strict syntax and semantics and supports Step-Wise Refinement Method. Two types of abstractions (called 'semantic identifiers') are used: operators and conditions, which can be elementary (atomic) and compound (composed from elementary by means of operations of Glushkov-Tseitlin Algebra of Algorithms like sequence, if-then-else, while-do and others).

Thus, in our example, only the following semantic identifiers:
"Analysis of the equation Ax2+Bx+C=0 ",
"Solution of the quadratic equation Ax2+Bx+C=0"
"Solution of the linear equation Bx+C=0"
are compound operators, whereas all the rest abstractions are elementary operators and conditions.

It is easy to see, that SAA-schema has multilevel, tree-like structure: each level is described by a "symbolic equality" and elementary operators and condition are the leaves of this tree.

Input of MULTIPROCESSIST System consisted of two files:

- SAA-schema file.
- L-implementations file.

(List of L-implementations of elementary semantic identifiers of SAA-schema, developed manually by a programmer. It has the following structure:

@ <Name of the elementary semantic identifier1 >
  <L-implementation of the semantic identifier1>
@ <Name of the elementary semantic identifier2 >
  <L-implementation of the semantic identifier2>
and so on).

Synthesis of a PROGRAM consists of two steps:

STEP 1: MULTIPROCESSIST automatically translates SAA-schema into **Formula** (formal representation of SAA-schema) and **Table of Correspondence** of SAA-schema and Formula identifiers:

Formula MMM;
B0 = **if** V0 **then** (B1) **else** (B2);
B1 = A0*A1*(**if** V1 **then** A2)*(**if** V2 **then** A3)*(**if** V3 **then** A4);
B2 = **if** V4 **then** A5 **else** (**if** V5 **then** A6 **else** A7);
end Formula MMM;

Table of Correspondence

Formula	**SAA-schema**
B0	"Analysis of the equation Ax^2+Bx+C=0 "
V0	' Coefficient A <> 0'
B1	"Solution of the quadratic equation Ax^2+Bx+C=0"
B2	"Solution of the linear equation Bx+C=0"
A0	"Calculate: Z=-B/2A"

and so on.

STEP 2: MULTIPROCESSIST synthesizes source code of L–program using **Formula** and L-implementations file.

COMMENTS

- Some details and intermediate steps, related to Formula Verification and PROGRAM Optimization, are omitted.
- Glushkov-Tsetlin Algebra of Algorithms is a universal algorithmic system and thereby SAA/1 is Universal Algorithmic Language.
- **Formula** is also a description in SAA/1 language.
- One of the main features of SAA/1, distinguishing it from any Programming or Domain-Specific Language, is its "Open at the Bottom" feature. It means that there are no restrictions on the choice and quantity of the primitives of SAA-schema - elementary operators and conditions. These primitives can be of any level of abstraction and can have any semantic interpretations. (By the way, the language of flow-charts also has this feature).
  Hence SAA/1 is an Algorithms Description Language of any Level of Abstraction.
- There is also a possibility for Automatic Program Synthesis, when L-implementations file is preset in advance and consists of functions, covering some strictly defined Application Domain (e.g. BANKING, ACCOUNTING etc.)

and for any SAA-schema the set of its primitives consists only of the semantic identifiers of these functions. In this case we get some kind of Domain Specific Language.

Software Understanding and Programming Languages

Programming Languages are still widely used for both Design and Construction (Coding). Unfortunately, all significant programming paradigms like Structured Programming, Functional Programming, Object-Oriented Programming and others, destined for Software Design, were embedded in Programming Languages, making them overcomplicated, hard for learning and efficient utilization, and worsening program code.

"Software in Science, Engineering and Business requires domain expertise. But software development is worsened by the fact that the description mechanisms - programming languages - are inadequate for conveying relevant information to people. As a result, software systems become fragile monuments of code that only software "artisans" dare to modify" [9].

"Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of the large number of conflicting objectives for each of our programming projects. If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution." [10].

We can easily make sure, that the more a Program is efficient as to execution parameters, the less it is understandable, and vice versa. In other words, Machine-Oriented Efficiency (Speed, Memory…) and Human-Oriented Efficiency (Understanding, Readability…) are conflicting objectives of Programming Languages:

Analysis of these and other factors of weak Program Understanding has led to the following fundamental conclusion [7, 8]:

The main reason of weak Program Understanding is Programming Language Destination.
The Destination of any Programming Language is EFFICIENT EXECUTION of Program Code.
Therefore weak Program Understanding is fundamental and ineradicable attribute of any Programming Language.

Hence methodological conclusion:

Design must be separated from Implementationand supported by languages of a new type, absolutely different from classical Programming Languages.

These languages - Software Design Description Languages – must be specially focused on Software Design and, at the same time, enable Automated Synthesis of Program Code.

Tools supporting Software Understanding

One of the reasons of weak Software Understanding is that there is a large gap between the initially posed problem, which requires creation of a new software-intensive system, and its eventual solution. The Web of Design Ideas i.e. the web of thoughts, conceptions, intentions, design alternatives etc.., supporting smooth transition from the Software System's Goal to Software System's Creation, is hidden in the minds of stakeholders of software projects, disseminated in the Program Code or simply lost, and this is a very serious reason of weak Software Understanding.

"The gap between problem and solution must therefore be chiefly bridged by the human activities of describing, reasoning, designing, communicating, analyzing, and inventing, and software tools are needed to support those human activities and, by doing so, to amplify our human powers. The sine qua non of those activities is human understanding. In large measure, then, enabling and supporting human understanding must be the goal of the software engineering tools" [1].

Design Thinking, Creative Thinking and Learning

Design Thinking, *Creative Thinking* and *Learning* are aspects of Human Intellectual Activity, which are very close to each other.

What is the result of *Creative Thinking* of a person? It can be defined as a solution of some problem. While solving a problem a person produces in his mind some set of abstract ideas.

Creative Thinking is Processing of Abstract Ideas and Generation of Productive Ideas by a Human Brain

Abstract Idea can be represented as a network (web) consisting of various notions (nodes) and relations (links) between notions.

The most important feature of Abstract Idea is that in the process of Creative Thinking it is being changed permanently.

Processing of Abstract Ideas means that they are being changed permanently in terms of such *intellectual operations* like *concretization* of notions, *generalization*, *substitution*, *deletion* and others i.e. some "more abstract" notions are being changed by their "meanings" (i.e. by subnets, containing "more concrete" notions), other subnets are being generalized in "more abstract" notions, some notions are deleted, others are created and so on.

"*Design thinking* is what people do when they pursue their goals… It is a process of creative and critical thinking that allows information and ideas to be organized, decisions to be made, situations to be improved, and knowledge to be gainedetc. *Design Thinking* is creative because its outcomes are not predetermined" (Charles Burnette, IDeSiGN curriculum [11]). *Design Thinking* is a creative process based around the "building up" of ideas" [12].

Design Thinking is Creative Search of the deep-laid simplicity in a complicated tangled problem.

Learning is a process which is very close to *Design Thinking*. The only distinction lies in the fact that Outcomes of *Learning* are already predetermined in contrast to Outcomes of Design.

All content to be learned must be intellectually constructed.

This very old idea goes back to the outstanding ancient Greek philosopher Plato and was independently reinvented by D.Norman [13].

Well-known Hypertext Learning Model is based on the following hypothesis:

Creative Thinking is being accomplished associatively.

A network of *Writer*'s ideas, represented in hypertext form, gives to the *Reader* direct access to this network. Productivity of work of the *Writer* and adequacy of perception of the material by the *Reader* essentially grows, because intermediate transformations of the material are excluded. Moreover, using *Authoring* and *Browsing* this network of ideas of the *Writer* can be effectively analyzed and transformed by the *Reader* into his own network with the account of not only the essence of information, but also of his individual mental and psycho-physiological features.

It is noticed, that *Browsing* stimulates *Creative Thinking* and in the process of *Browsing* an Effect of Serendipity can appear. Moreover, there is the following hypothesis of D. Engelbart [14]:

An opportunity to manipulate external symbols generates an effect of Augmentation of the Human Intelligence.

Tete-A-Tete_Hypertext

One of the technological outcomes of Tete-A-Tete is Interactive Hypertext Technology (Tete-A-Tete_Hypertext) -creativity enhancing tools based on innovative hypertext technology supporting all the intellectual operations mentioned above [15,16].

Tete-A-Tete_Hypertext is oriented above all on computer-aided support of Creative Thinking in contrast to conventional text processors like MS WORD or HTML-editors, oriented on preparation of printed documents.

Tete-A-Tete_Hypertext is based on Interactive Hypertext Model (versus Static Hypertext Model used in the majority of hypertext technologies like help-systems of Windows-programs or HTML-files):

- *Static Hypertext Model: Hypertext structure is fixed at the design stage and user can only accomplish Browsing.*
- *Interactive Hypertext Model: Hypertext structure transformations are being accomplished ONLINE.*

The Tete-A-Tete_Hypertext motto (and, at the same time, the goal of its development) is: *What-You-THINK-Is-What-You-Get.*

Mathematics for Software Design

Analyzing applications of classical Mathematics to Software Design (Set Theory, Category Theory, First-Order Predicate Calculus and others), We came to the following fundamental conclusion:

Mathematics, based on Set-Theoretical Paradigm, is not an adequate tool for Software Design.

Set-Theoretical Paradigm

- The Set exists because its Elements exist.
- The Set is uniquely defined by its Elements.
- Elements are primary, the Set is secondary.

In contrast to Set-Theoretical Paradigm there exists System-Theoretical Paradigm:

System-Theoretical Paradigm

- The System is the Whole, the Integrity of something.
- The System can be represented in many different ways as a structure consisting of interrelated and interdependent Elements (Syn: Components, Entities, Factors, Members, and Parts etc.).
- The System is primary and Elements are secondary.

Author is deeply convinced that

Mathematics for Software Design must be based on System-Theoretical Paradigm.

Algebra of Abstract Systems [17] is the first steps in the process of creating System-Theoretical Mathematics.

Algebra of Abstract Systems is based on "Z-multiset" notion: a set together with integer multiplicities of its elements. This notion is a generalization of such notions as "set" and "multiset" (a set with nonnegative multiplicities of its elements, used in Petri Nets and Combinatorial Analysis).

Algebra of Abstract Systems consists of elements (e.g. concepts, events, states and others) and basic operations on elements, such as addition/subtraction, concatenation (sequence), multiplication (vectorization), non-deterministic choice (one-of), interrelation, specialization, substitution, conditional existence (if-then), and some other operations.

There are elementary (atomic) elements and compound elements (composed from elementary by means of operations of *Algebra of Abstract Systems*, and there are derived (composed from basic) operations.

Very important feature of this Algebra is that these operations have natural algebraic properties like associativity, distributivity and others, thereby enabling to describe and transform various systems as algebraic formulas in a very succinct and clear manner.

Universal Software Design Description Language

The main novelty of Tete-A-Tete is a universal Software Design Description Language (*Tete-A-Tete_Language*).

Tete-A-Tete_Language should:

- be absolutely independent from any Implementation Environment (Operating Systems, Programming Languages, Scripting Languages, Databases etc.).
- be a controlled natural language, which is easier to understand. A controlled natural language is a computer processable subset of standard English designed to serve as knowledge representation language. It will be based on some ideas of the Attempto Controlled English (computer processable subset of standard English [18-19]) and Semantics of Business Vocabulary and Business Rules (SBVR) [20].
- be a kind of expressive and powerful formal language, based on some new Mathematics specially oriented on Software Design.
- have "*open at the bottom*" feature, distinguishing it from any computer language. "*Open at the bottom*" means that language primitives are not settled i.e. there are no restrictions on the choice, quantity and interpretation of language primitives. This feature implies that Tete-A-Tete_Language should be a universal formal language of any level of abstraction.
- It will provide smooth transition from informal Problem Domain descriptions in the human language (using Tete-A-Tete_Hypertext) to more and more formal descriptions (in terms of abstract Concepts, Relations, Scenarios, GUI Controls, etc.), which will be insofar compiled for appropriate target Implementation Environment.

Software Director - a new stakeholder of Software Development

Software Director should be:

- the Chief Architect of the Software System and the Main User of Tete-A-Tete_Language.
- solely responsible for the Conceptual Integrity of the Software System [2] and unifying Software Development under the vision of a single person.
- someone in a Software Development Process analogous to the Theatre Director or Stage Director (Régisseur) in the theatre field, who oversees and orchestrates the mounting of a theatre production.
- the only mediator between non-programming stakeholders (Customers, Subject Matter Experts, and End-Users …) from one hand, and professionals in Software Construction (Programmers, GUI Designers, QA Engineers, Testers and others) from another hand.

Tete-A-Tete_Designer automated software development tools

Tete-A-Tete_Designer should:

- be based on Tete-A-Tete_Hypertext;
- support Tete-A-Tete_Language;
- enable Automated Generation of up to 60-75% lines of Program Code as well as Automated Generation of Database Schemas and User Acceptance Tests for various Implementation Environments.

We consider a software system as a partially ordered set of **tasks** that are implemented by **scenarios**. Each **scenario** is a sequence of **cadres (frames)**. Each **cadre** is a primary logically complete element of the software system under design consisting of a **screen form** (Windows Form or Web Page), the **user's actions** on the form and the **system's reactions** on events which may happen on this form.

Examples of Descriptions in Tete-A-Tete_Language

Problem Domain Concepts

```
concept   USER_PROFILE     is-defined-as sum-of
                FIRST_NAME
                LAST_NAME
   concept      DATE_OF_BIRTH
   concept      GENDER
   concept      ETHNICITY
   concept      RACE
   concept      ADDRESS
                HOME_PHONE_NUMBER
                MOBILE_ PHONE_NUMBER
   concept      EMAIL
end-of-concept USER_PROFILE

   concept DATE is-defined-as product-of
                MONTH
                DAY
                YEAR
   where
   concept YEAR is-defined-as
                 digital-word year such-that
                length-of year is-equal-to 4
   end-of-concept YEAR

   concept MONTH is-defined-as word month
        such-that month is one-of
        "January" "February" "March" "April" "May" "June" "July" "August" "September" "October" "November" "De-
        cember"
   end-of-concept MONTH

   concept   DAY is-defined-as digital-word day
        such-that
        either
        day is-equal-to one-of 1 2…30
        in-case month is-equal-to one-of
        "April" "June" "September" "November"
        or
        day is-equal-to one-of 1 2…31
        in-case month is-equal-to   one-of
        "January" "March" "May" "July" "August" "October" "December"
        or
        day is-equal-to 29
        in-case month is-equal-to   "February"
                and
        year is-a LEAP_YEAR
        or
        day is-equal-to 28
        in-case month is-equal-to   "February"
        and
        year is-not LEAP_YEAR
    end-of-concept DAY
    end-of-concept DATE
```

User-System Dialog Interfaces

```
    form USER_PROFILE is-defined-as sum-of
        input     FIRST_NAME
        input     LAST_NAME
        form      DATE_OF_BIRTH
        selection          GENDER
        selection          ETHNICITY
        form      RACE
        form      ADDRESS
        input     HOME_PHONE_NUMBER
        input     MOBILE_ PHONE_NUMBER optional
        input              EMAIL
        button    "Submit"
    end-of-form   USER_PROFILE


    form DATE OF BIRTH is-defined-as product-of
        selection          MONTH
         selection         DAY
         selection         YEAR
    end-of-form   DATE OF BIRTH

    form RACE   is-defined-as sum-of
        option    "African American"
        option    "Asian"
        option    "Caucasian"
        option    "Hispanic"
    end-of-form   RACE
```

User-System Dialog Scenarios

```
scenario-for-task "User Registration" is-defined-as

        USER should start-application

        SYSTEM should check-task-feasibility

        DIALOG-INTERFACE should-be
                form "Home" containing
                button "Create Account"

        USER should "start Registration" defined-as
                click-button "Create Account"

        SYSTEM should show-form "Registration"

        DIALOG INTERFACE should-be
                form   "Registration" containing
                form    “Account”
                button  “Register”
        where
        form  “Account” is-defined-as sum-of
                input   USERNAME
                hidden-input-with-confirmation                  PASSWORD
        input-with-confirmation EMAIL

USER should
        * identify-form  “Registration”
        * “create new ACCOUNT” defined-as
                fill-in-form “Account”
                click-button “Register”

SYSTEM should
         “check and save ACCOUNT “ defined-as
                if form  “Registration” is-valid
                and      LOGIN is-unique
                then   “accomplish User Registration”
                else   show-form “Registration” errors
where “accomplish User Registration” is-defined-as
                save     ACCOUNT
                show-form “Successful Registration”

DIALOG-INTERFACE should-be
        form “Successful Registration” containing
                link “Confirm Registration”

USER should
        * identify-form “Successful Registration”
        * “finalize User Registration” defined-as
                click-link “Confirm Registration”

SYSTEM should
        set-task-successful-termination
end-of-scenario-for-task “User Registration”
```

Related Work

There are a few approaches similar to Tete-A-Tete:

1. Literate programming [21]:

"A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code.

In literate programming the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code. No longer does the English commentary injected into a program have to be hidden in comment delimiters at the top of the file, or under procedure headings, or at the end of lines. Instead, it is wrenched into the daylight and made the main focus. The "program" then becomes primarily a document directed at humans, with the code being herded between "code delimiters" from where it can be extracted and shuffled out sideways to the language system by literate programming tools.

The effect of this simple shift of emphasis can be so profound as to change one's whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick." [22].

2. Intentional Programming [23]:

"Businesses invest a great deal of time and expense developing software. But all too often the knowledge and insights gained during the development disappear into the details of the code or at best only exist in documents with slender ties to the actual source code. Another name for this latent value is the intent behind the software. That is why we call this approach Intentional Software™.

Intentional Software captures the tremendous latent value that is usually lost in the design and development process and makes it part of the software. Using Intentional Software the domain knowledge is captured not lost. All stakeholders - programmers, domain experts and others - can have their design intent clearly represented in code. "[24].

Conclusion

Implementation of the Tete-A-Tete Project should:

- have broader impact on Software Reengineering making it much easier, because it will become Continued Development in the Tete-A-Tete_Designer:

Software Project, specified in the Tete-A-Tete_Language and independent from any Implementation Environment, can be recompiled (after all the required transformations) for any Implementation Environment, available in Tete-A-Tete_Designer.

- lead to reinvention of the Chief Programmer Team Management approach [3, 4]:

Chief Programmer Team should be replaced with Software Director Team. Software Director should be equipped with the Tete-A-Tete_Designer and thereby the duties of other Software Director Team members (Developers, Testers, GUI Designers…) should be substantially subordinated to Software Director, enabling him to unify Software Development under the vision of a single person in order to achieve Conceptual Integrity of the Software System.

1. *Jackson M.* Automated Software Engineering: Supporting Understanding. Automated Software Engineering. – 2008. – Vol. 15, N. 3. – P. 275–281.
2. *Brooks F*. The Mythical Man-Month, Addison-Wesley, first published 1975. Silver Anniversary edition published 1995.
3. *Mills H. D*. Chief Programmer Teams: Principles and Procedures, Report N. FSC 715–108, IBM Corporation, Gaithersburg, Maryland, June, 1972.
4. *Baker F. T*. "Chief Programmer Team Management of Production Programming" // IBM Systems J. – 1972. – Vol. 11, N. 1. – P. 56–73.
5. *Yuschenko E.L., Tseitlin G.E., Hrytsay V.P., Terzjan T.K.* Multilevel Structured Program Design: Theoretical fundaments, tools. – M.: Finances and Statistics, 1989. – 208 p. (in Russian).
6. *Hrytsay V.P., Tseitlin G.E.* Some questions of Structured Parallel Programming Automation // Cybernetics. – 1979. – N l. – P. 106 – 111 (in Russian).
7. *Hrytsay V.P.* On Implementation of MULTIPROCESSIST Structured Programming Tool. Cybernetics. – 1983. – N 3. – P. 118–123 (in Russian).
8. *Hrytsay V.P.* Design Principles and Implementation of MULTIPROGESSIST Structured Programming Tool. Ph.D. Thesis. Kiev. –1986 (in Russian).
9. *Lopes C.* A Linguistic Approach to Software Development, NSF Award Abstract #0347902, http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0347902.
10. *C.A.R. Hoare* (1981). The emperor's old clothes". Communications of the ACM 24 (2): 5–83.
11. http://www.idesignthinking.com/01whyteach/01whyteach.html
12. http://en.wikipedia.org/wiki/Design_thinking.
13. *Norman, Donald A.* Defending Human Attributes in the Age of the Machine, First person CD-ROM, New York, Voyager, 1995.
14. *Engelbart D.* A conceptual framework for the augmentation of man's intellect. In P.W.Howerton and D.C. Weeks (Eds).// Vistas in information handling: The augmentation of man's intellect by machine.-Washington, DS: Spartan Books. – 1963. – 1. – P. 1–29.
15. *Hrytsay V.P*. "Tete – A – Tete Dynamic Hypertext" information technology. Herald International Solomon University. – 2000. – N. 4. – P. 124–133 (in Russian).
16. *Hrytsay V.P*. Tete – A – Tete Hypertext Technology – a tool for development of Computer – Aided Learning Intellectualization Systems. USiM. – 2002. – N. 3/4. – P .87–91 (in Russian).
17. *Grits'ay V.P*. Algebra "GAMMA" of abstract systems: a mathematical background of a parallel program development specification language. IMACS 14 World Congress, Proc. IMACS 14, Atlanta, USA. – 1994. – Vol. I. – P. 200–202.
18. *. N. E. Fuchs, U. Schwertel, R. Schwitter*. Attempto Controlled English (ACE) Language Manual, Version 3.0, Technical Report 99.03, Department of Computer Science, University of Zurich, August 1999.
19. *Stefan Hoefler*. The Syntax of Attempto Controlled English: An Abstract Grammar for ACE 4.0, Technical Report ifi-2004.03, Department of Informatics, University of Zurich, 2004.
20. http://www.omg.org/news/meetings/ThinkTank/past-events/2006/presentations/04-WS1-2_Hall.pdf
21. *Donald E. Knuth*. Literate programming // The Computer Journal, May 1984. – 27(2):97111.
22. http://www.literateprogramming.com
23. *Charles Simonyi*. The Death Of Computer Languages, The Birth of Intentional Programming, September 1995 Technical Report MSR–TR–95–52.
24. http://www.intentsoft.com/technology/overview.html