УДК 378.147:372.862:004, ВАК 05.13.01, ГРНТИ 28.01.45

УРУСОВ Н.
ОБУЧЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ ПО АЛЬТЕРНАТИВНОЙ УЧЕБНОЙ ПРОГРАММЕ: ИДЕИ И МЕТОДЫ МАЙКЛА А. ДЖЕКСОНА В УХТИНСКОМ ГОСУДАРСТВЕННОМ ТЕХНИЧЕСКОМ УНИВЕРСИТЕТЕ, РОССИЯ (С 2014 ПО НАСТОЯЩЕЕ ВРЕМЯ)

Обучение программной инженерии по альтернативной учебной программе: идеи и методы Майкла А. Джексона в Ухтинском государственном техническом университете, Россия (с 2014 по настоящее время)

Н. Урусов

Нью-Лондон, США

Данная статья направлена в поддержку применения идей и методов Майкла Джексона в университетском учебном курсе по разработке программного обеспечения. В нем содержится краткое описание методов разработки систем и структурного программирования Джексона, а также фреймового представления задач, в том числе его существенного уточнения в ближайшей перспективе. Далее в работе обсуждается опыт автора по преподаванию курса "Введение в программную инженерию: идеи и методы Майкла Джексона" в течение последних трёх лет, начиная с 2014 г. в Ухтинском государственном техническом университете, Республика Коми, Россия, и приводятся извлечённые уроки и планы будущей работы.

Ключевые слова: *программное обеспечение инженерное образование, дистанционное обучение, методы разработки программного обеспечения,*

Teaching an Alternative Software Engineering Curriculum: The ideas and methods of Michael A. Jackson at Ukhta State Technical University, Russia (2014 – Present)

N. Ourusoff

New London, USA

This paper is an apology for focusing on the ideas and methods of Michael Jackson in university software engineering curricula. It contains a brief summary of the Jackson structured programming and Jackson system development methods, and the Problem Frames approach, including a significant refinement to Problem Frames in a draft chapter of a forthcoming work. The paper then discusses the author's experience teaching a course: "Introduction to Software Engineering: The Ideas and Methods of Michael Jackson" the past three autumns beginning in 2014 at Ukhta State Technical University, Komi Republic, Russia, and concludes with lessons learned and modest plans for future work.

Keywords: *Software engineering education, Distance learning, Software development methods, Design patterns, Embedded and cyber-*

шаблоны проектирования, встроенные системы, надёжность программного обеспечения, юзабилити, отказоустойчивость, анализ требований

physical systems, Software reliability, usability, fault tolerance, Requirements analysis

1. Introduction

Previously, I have described using the ideas and methods of Michael Jackson as the basis of an alternative curriculum to teaching a 1st course in Software Engineering [25]. Here, I wish to report on further efforts to disseminate and develop this curriculum at Ukhta State Technical University, Komi Republic, Russia during autumn semesters since 2014.

2. Why Jackson's Ideas and Methods?

Over more than four decades – from the early days of computing in the early 1960's to the present – Jackson has contributed four books; over 100 articles; and three original, sound methods or approaches to software development – JSP, JSD (with John Cameron) and Problem Frames[3]; he is on the Editorial Board of four leading international Software Engineering publications, and a member of two IFIP working groups. He has received several research awards, including the IEE Achievement Medal, The British Computer Society Lovelace Medal, and the ACM Sigsoft Outstanding Research Award[4].

Jackson's ideas and methods are sound, original, and have evolved as the cutting-edge of research and practice for more than four decades: they deserve to be incorporated into the software engineering curriculum [25]. For an excellent summary of JSP, JSD and Problem Frames, the reader is referred to [7, 9, and 10 respectively]; we briefly summarize the JSP and JSD methods and the Problem Frames approach, and remark that since the publication of *Problem Frames* (2001) more than 15 years ago, Jackson has been clarifying and redefining ideas in software development about: requirements, behavior, and goals; the role of top-down and bottom-up design, and the role of formalism and intuition. In a draft chapter of his forthcoming book [10], Jackson declares *system behavior* as the core object of software system development and introduces a method for representing problem decomposition as an *intelligible structure* of system behavior – a rooted tree of instantiations of its *constituent behaviors*. His forthcoming book promises a significant refinement to the Problem Frames approach to system development.

Michael Jackson first became well-known as a result of his *program design method*, Jackson Structured Programming (JSP), which emerged from the early days of commercial computing in the early 1960's, and culminated in his classic, *Princi-*

[3] For Jackson, a method is a step-by-step decision-procedure. JSP and JSD are methods; Problem Frames is an intellectual *approach* to analysis and structuring of software systems.
[4] http://fose.ethz.ch/speakers.html

ples of Program Design (1975). His design method is applicable to a class of programs common in batch processing systems[5] prevalent during the 1960's to which Jackson gave the name *simple programs:* these are programs that can process a set of related *sequential data streams* in a single *program structure of corresponding components*. JSP is very likely the only program design method that became a national standard [18]; has been described as the best design method of the time [1]; and has been certified in an Associated for Computing Machinery (ACM) column as producing the correct program design [27].

JSP is based on the control structures of structured programming. But, while goto-less programming gives a well-formed (spaghetti-less) structure, it does not guarantee an *intelligible design*. The main idea of JSP is very different from modular programming, the prevailing design approach during the 1970's, which focused on decomposition of a program into separately compiled modules to achieve coherence. [Jackson, 7] In JSP, a program's structure is formed from the structure of its input and output data streams. In contrast to top-down decomposition, the then prevalent "best-practice", JSP is a *constructive method*[6] for composing a program's input and output data structures into a composite program structure containing *corresponding components*. Design is about *structure:* A program structure becomes *intelligible* – is well designed – if its structure relates directly to the problem world. Jackson introduced structure diagrams – later called *tree diagrams* – to depict program structure, and contrasted them with flow charts, which were in vogue, but which, as he pointed out, show a program's (dynamic) flow of control, not its (static) structure.

In JSP, Jackson formulated several program *design patterns* as rules[7], especially concerning the placement of read-write operations that were a frequent source of error in the magnetic tape-based business-oriented batch processing systems of that period. There is a *read-ahead rule* which applies to situations in which the condition for processing input became immediately available:

"Place the initial read immediately after opening a file, prior to any component that uses a record; place subsequent reads in the component that processes a record, immediately after the record has been processed" [5].

A *multiple read-ahead rule* applies to situations in which the condition for processing input can be *recognized* only after a fixed number of read operations; and *backtracking*[8] applies if the recognition cannot be resolved by any predetermined number of read operations.

[5] In batch processing systems, in contrast to systems that process transactions one-at-a time as they become available, transactions are accumulated into batches and sorted into the sequence of the primary key of the entity to which they apply before they are processed. To process each transaction as it arrives, random access storage is generally necessary, but was not yet economically viable until the 1970's.

[6] By method, Jackson means a *step-by-step decision procedure.*

[7] These rules are associated with a form of control structure commonly used in which the condition occurs at the beginning, as in: do while<condition>… endwhile

[8] Jackson added *backtracking,* as a 4th control structure in Jackson structured programming. It is a restricted form of selection, that resolved recognition problems in the case when no finite number of read-aheads could resolve a parsing problem. The semantics of backtracking are close to Java's *try…catch*.

The Basic JSP method can be extended if a *structure clash* – the absence of correspondence in the structures of a pair of (input, output) files – is encountered. Structure clashes prevent the construction of a single program structure as required by the basic JSP method. By *decomposing* a program into two simpler programs, the structure clash can often be eliminated: the first program writes an intermediate file – without any structure clash – that the 2nd program can process; so that, both programs can be designed with JSP. The additional complication of writing and then reading an intermediate file can be eliminated through the use of Jackson's *program inversion*. A program that writes a file that a 2nd program reads, can be algorithmically transformed into a single-entry, multiple-state subroutine that does not write its output stream directly but passes the next output record to the 2nd program. We say that the program is *inverted* with respect to its output file.[9] The communication between the program and subroutine is immediate, facilitating efficient scheduling. More important, an inverted program is structurally the same as the program from which it is derived, and its subroutine representation can carry variable state just as does a program.

Because of program inversion (see below), JSP's range has been extended far beyond the programs written for systems prevalent in the 1970's to include interactive systems, interrupt handlers, Web database queries, embedded systems and handling network protocols. The class of simple programs fits one of 5 elementary problem frames described in *Problem Frames,* and JSP is an appropriate method to use in problems of this type.

Jackson's 2nd major contribution to software method, developed in conjunction with John Cameron, is Jackson System Development (JSD) [5]. Top-down hierarchical functional design was the prevailing "best practice" approach to systems design during the 1970's and early 1980's. JSD design was different: JSD is based on an *analogical* (simulative) *model*, not functions. The abstract models are of real-world *entities*[10] connected via serial data streams to form a network of *model processes*, each exhibiting their time-ordered behavior. A JSD system is thus entity-based (entities are somewhat analogous to objects in OOP[11]) and has been termed "middle-out" [2] in contrast to "top-down". The scope of a model is deemed to be sufficient to encompass a variety of functions that may be required – the functions can be added later to the network model. JSD is limited to real-time, systems.

Both JSP and JSD process sequential streams using the control structures of structured programming. Two ideas from JSP connect JSP to JSD:. The first idea is

[9] Similarly, a program that reads a file that a 1st program writes, can be algorithmically converted into a single-entry, multiple-state subroutine that does not read its input stream directly but receives the next record from the 1st program. We say that the program is *inverted* with respect to its input file.

[10] Entities have a long philosophical history, where they aimed to clarify the essence of 'beings' (things or objects) in the world. Both JSD and OOP model real-world objects, but Jackson holds that the real-world is informal, while in OOP, objects are computational and formal..

[11] Simula was the 1st object-oriented programming language, developed in the 1960's; Simula67 is regarded as the 1st OO language. Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67.[21] Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of *Byte Magazine*. [Wikipedia]

the *tree diagram* to represent structure.. The time-ordered behavior of entities can be depicted with *entity structure diagrams* that are isomorphic to *program tree diagrams (*entity substitutes for program; and entity action for program operation*):* so, JSP is a natural fit as a design method for JSD systems.

The second idea that connects JSD to JSP is *program inversion.* When a transaction record (in a transaction processing system) is encountered, the *program text* of the denoted entity type, represented as a long-running, interruptible multiple-state subroutine, is activated (resumed) and *consumes* the transaction, updating the state and data of its state vector, before becoming inactive (until the next transaction arrives). Just as the static tree structure of a program in JSP traces the program's behavior, so the static entity structure of its subprogram text traces the time-ordered behavior of individual entities.

Jackson's *Problem Frames* (2001), grew out of work over a 15-year period that culminated first in a short book, *Software Requirements & Specifications* (1996). Jackson's research drew from practical consulting and from telecommunications modeling for AT&T: His short book is a clarification, redefinition, critique, and invention of terms in software engineering.

As a result, *Problem Frames* is an original departure from the more than two decades of current "best-practice" using various forms of Object-oriented Analysis and Development (OOAD). Jackson's approach to software development is *problem oriented* rather than *solution oriented*: Analysis initially involves considerable informal exploration. A context diagram shows where a problem is located in the world, and consists of three elements: problem *domains* – where the problem is located; *interfaces* – consisting of sets of phenomena shared among domains; and the *machine, a* special domain consisting of the computer and the software to solve the problem. A *problem diagram* adds the *requirement* – representing stakeholders desires and wishes together with collaborative input from developers – that a machine must satisfy. The heart of analysis is three *separate development descriptions:* the machine specification; characteristics (properties) of the problem domains; and the requirement. To convince the customer or stakeholder that a system will *behave* as required, we must show that the specification (S) together with the domain properties of the problem world (P) satisfy the requirement (R): S, P |= R.

There is almost always a need to decompose a realistic problem into two or more sub-problems. The initial decomposition is top-down, is usually guided by *intuition*, and hopefully results in sub-problems that are simpler and that either fit – or will fit after further decomposition(s) – one of *five elementary problem frames* or problem patterns that Jackson has discovered to be prevalent in all or most software applications. Jackson had emphasized that systems are structured as layers that are run in parallel [4], rather than as a hierarchical *partition.* In Problem Frames, this ideas is re-formulated more precisely: sub-problems are *projections* of the whole problem, analogous to projection in the relational model. Just as the same column(s) may appear in two or more different relational projections, so too in Problem Frames, the same domain(s) may exist in two or more sub-problems. Each sub-problem is treated as a separate problem – deferring consideration of interactions with other sub-

problems until it is well understood – and analysis proceeds with separate development descriptions for its machine specification, problem domain properties and requirement. Only during implementation – implementation is not treated in *Problem Frames* – is the problem of how to run the sub-problem machines in parallel addressed.

In the Problem Frames approach, Jackson stresses that five *special concerns* must almost always be addressed: overrun, initialization, completeness, reliability, and identities. Each frame is vulnerable to different special concerns.

Jackson believes that use of problem frames to structure the analysis and decomposition of problems into elementary problem frames may result in a *standard (normal) practice* in software engineering – a practice that is accepted and supported by software engineering professionals as evidenced in the professional literature and meetings over a period of several years. Jackson argues that *normal practice* has been shown to be necessary in other engineering sciences, but is lacking so far in software engineering. So, in his view, at the present time, software engineering is not yet a science.

An overriding structuring principle throughout problem analysis is *separation of concerns:* An example of it is the need for three separate *development descriptions.* The specification, domain characteristics and the requirement of a system represent different viewpoints (stakeholder, requirements analyst, and programmer, respectively) and express different linguistic modalities (we *hope* the specification is correct; we *wish* for the requirement to be satisfied: *optative;* the properties of the real-world domain are based on laws of physics: *indicative*. Because of these differences, we need to describe each separately – otherwise we get a confusing description. Another example is in analysis: we analyze each sub-problem *separately*, ignoring interactions with other sub-problems until we fully understanding it in isolation, and return to consider its interactions later.

Since the publication of *Problem Frames*, Jackson has focused on a number of questions (in ~40 published articles)[12] including:

1. What are requirements? How do requirements, behavior and the goals of a system differ?

2. What is the core artifact of system development?

3. How can we make cyber-physical systems i*ntelligible*?

4. When are top-down and bottom-up reasoning called upon during system development?

5. What is the appropriate role for intuition and formalism in the course of system development; and when should each be used?

Jackson argued that objects are too fine a granularity as the basis for software development; that reliance on use cases is solution-oriented, whereas problems are further away from the machine and require focused analysis; and that OOP objects are a formal straight-jacket on real-world entities [5]. OOP objects are static – they cannot change type; nor is multiple-inheritance supported, whereas, in the real world,

[12] http://mcs.open.ac.uk/mj665/papers.html

living objects undergo metamorphosis and support multiple inheritance. With respect to OOAD's reliance on use cases as the basis for constructing a system's behavior, there are few if any use cases in continuous-state monitoring systems.

As Jackson increasingly turned his attention to complex, real systems, he began using the term, *cyber-physical systems*, which refer to the integration of computation with physical processes.[13] In cyber-physical systems, a software machine interacts with a human and material problem world, monitoring and controlling its behavior to ensure that the many and varied system requirements are satisfied as fully as possible. The development of cyber-physical systems involves *diverse and complex* tasks, each of which require different descriptions and methods. It is the *complexity* of cyber-physical systems, that leads Jackson to conclude that no single technique or system (formal language or logic) can be applied from start to finish to make this complexity intelligible.

As shown in a draft chapter [11] of his forthcoming book, *Behaviours as Design Components of Cyber-Physical Systems*, Jackson's focus on *systems behavior* and its representation as a *behavior control tree*, represents a significant refinement to the Problem Frames approach to designing cyber-physical systems.

The approach to analysis in Cyber-Physical systems deals with the pre-formal work of creating a bridge between the stakeholders' purposes and desires – the *requirement*; and a detailed *system behavior* agreed upon by stakeholders and system developers and expressed in natural language.

The core object of systems development is *system behavior*. The behavior of a cyber-physical system is viewed as a set of heterogeneous behaviors, each of which can be decomposed (top-down) into *constituent simpler behaviors*. Each must be designed separately, following the *separation of concerns* principle. For each behavior, a *machine* (specification) must be designed (bottom-up) that ensures the system's behavior. Jackson has invented a method for rooting instances of these constituent behaviors in a single dynamic tree, providing for cyber-physical systems the kind of *intelligibility* that is provided by well-designed programs using structured programming: a visible correlation between what is happening in the system – the execution of an instance of a constituent behavior – and its program text in the machine specification.

In the beginning of a project, the developer explores and describes the problem world. This exploration requires the ability to introduce new names, formal definitions, and informal denotations (recognition rules for states or events), to represent phenomena, and to continuously check their accuracy. The developer must be guided at first, not by any formal system, but by *intuition:* By intuition, Jackson does not mean an uncontrolled impulse to baseless conjecture and unfounded assertion, but rather the faculty of recognition and understanding on the basis of our accumulated experience, insight and knowledge, with little or no appeal to conscious reasoning. We use *natural language* which mediates between phenomena in the real world and open-ended reasoning.

[13] Wikipdia. The term, Cyber-Physical System was introduced *circa* 2010 by Helen Gill. Helen Gill, CISE Point of Contact for the CPS program, telephone: (703) 292-7834, email: hgill@nsf.gov.

Formal systems are extremely important once projections of the system behavior into constituent behaviors have been identified:

"...formalism has its proper place. Its place is not in the early stages of exploration and learning, where it is premature and restrictive, but in the later stages, where we need to validate our informal discoveries, designs and inferences by submitting them to the rigour of formal proof" [11]

3. Experience at Ukhta State Technical University

I plan to teach for the 4th time for the Department of Computer Science and Information System Technologies at Ukhta State Technical University during the autumn semester 2017. I am grateful to have been given the freedom to focus on my interest in Jackson's ideas and methods in software engineering; I also teach a second course, a short course in Object-oriented Design using Java to students who have had a previous course in a programming language (Pascal/Delphi). The second course is relevant to teaching Jackson's ideas and methods: First, Jackson's JSD method is entity-based, and entities are "objects", and thus have some common ground with OOP objects (though with important differences). Second, although Jackson's approach to software development is language independent, I have been using Java to illustrate program implementation in JSP and JSD; and Jackson frequently uses OOP object diagrams (among other notations) in *Problem Frames.*

3.1. Courses focusing on Jackson's ideas and methods (in chronological sequence)

1. An Intensive Short-course (Autumn 2014)

My 1st effort came to fruition from an invitation from Ukhta State Technical University to teach and consult on curriculum through the Fulbright Specialist program[14]. I gave approximately 14 lectures during 4 weeks, each consisting of 2 90-minute lectures,, a total of approximately 40 hours in all. The lecture content consisted of: JSP – 10 hours; JSD – 3 hours; Problem Frames – 27 hours. I lectured in English, and many of the students weren't able to follow the lectures or the slides on which they were based. Fortunately, the Chair of the Department, Dr. Felix Marakasov, offered to attend my lectures: This had two salutary effects: (1) it boosted attendance sharply, thus saving the course; and (2) every 20 minutes or so, he paraphrased the material that I had just presented into Russian, increasing student understanding and attention significantly. We covered the 1st 5 (of 12) Chapters in *Problem Frames*, and followed with a brief discussion of Chapter 7 (Model Domains) and Chapter 9 (Particular Concerns). In addition to the lectures, there were 4 Learning Tasks (practical exercises) in JSP, 1 in JSD, and 4 in Problem Frames; at the conclusion of the course, each student was given an an oral examination, based on 25 questions posted on the course Web site, with 1 question randomly selected from the JSP/JSD set of questions and 2 questions randomly selected from the set of Problem

[14] http://www.cies.org/program/fulbright-specialist-program

Frames questions[15]. At the end of this short course, a Certificate was awarded to each student who completed the course. Comments, but no grade, were given to Learning Tasks submitted by students.

2. Full-semester course (Autumn 2015)

The following autumn the same content was taught over 14 instead of 4 weeks, with the final test and oral exam given the 1st week of January. Each week there was 1 lecture (90 minutes); and 1 practice session (90 minutes) for each of 4 different groups: 2nd-year and 4th-year students in two disciplines: Information Systems and Technologies (IST) and Computer Science and Engineering (IVT). The groups were assigned team projects, with 3–5 students/team, and asked to develop a project definition and separate development descriptions for each problem and sub-problem, together with an associated context diagram and problem diagram. A small bi-lingual Question Data Bank was developed on the on the University's Center for Distance Education's (CDE) Moodle server and used to administer a final test for all students. Oral examinations – and a grade – were given to the IST students, while IVT students took the course as Pass/Fail. Student grades in the written and oral tests were used (together with submitted learning tasks and course participation) to give an overall grade (<3 – Failure, 3 = Satisfactory, 4 = Good, 5 = Excellent).

3. Distance-education course (Autumn 2016)

The same content was presented as in 2015, but after the 1st 4 weeks (covering JSP and JSD), the remainder of the course was taught at a distance, using the Moodle server and included weekly Webinars[16] until I returned for 2 weeks at the end to advise on student projects and administer grading. An end-of-semester test was given to all students, and an oral examination (similar to that given 2015, but with an expanded set of Problem Frames questions) to all groups except 4th year IST students, who took the course as Pass/Fail. Grades were determined, as they were for the 2015 course.

Projects were assigned for 4th-year students only. They could either document and implement one of four simple systems fully analyzed by Jackson in the 5th chapter of *Problem Frames* to illustrate analysis of elementary Problem Frames; or, they could define their own systems and provide analysis (i. e., development descriptions and their associated context and problem diagrams). Not surprisingly – good programmers just love to program! – 4 teams implemented the already – analyzed examples (using Java), and just 2 chose to define and analyze their own systems. I expected this result and knew that it was really an inappropriate assignment: the course was all about analysis, not implementation! However, I hoped that students would be happy programming a well-defined and analyzed problem – and at the same time, would see clearly the connection between analysis and implementation. They were asked to include development descriptions with diagrams as well as their programs and test results.

[15] Se Appendix 1 for the syllabus, and Appendix 2 for oral examination questions.

[16] A Webinar is an Internet classroom. On the CDE Moodle server, Webinars are enabled by BigBlueButton [http://bigbluebutton.org/], software that is integrated into the Moodle course management system, and provides presentations, video lectures, whiteboard, chat, and other features.

3.2 Lessons Learned

As a teacher, I confess to being more interested in what I am teaching than with how best to teach, and this is likely why I learn lessons slowly. Nevertheless, I present a few lessons from my teaching – whether I have really learned them will be evident in the next iteration.

1. Interacting with students by Webinars

First, an acknowledgment: I received and would like to acknowledge help and advice given to me by one of the specialists at the Center for Distance Education (CDE), Natalya Vassilyevna Soldatova., who scheduled the Webinars and was usually present in the CDE auditorium to give my students and me help with the BigBlue Button facilities, that allowed us to interact directly each week at a distance. I also received valuable help from Irina Marakasova, formerly Director of CDE.

In my first Webinar, I failed to get students to respond when I asked a question – they were reticent to go to the microphone in the CDE classroom where they met, probably due to their timidity to speak in English. CDE Specialist Soldatova suggested that I encourage them to "chat" – and that yielded much better interaction subsequently. She later showed me a useful Question facility in Moodle that allows students to link to an online test and take it during the on-line session. I used the Question feature as a way for students to learn during the Webinar, and only secondarily as a way to determine what they had learned. For example, I would create a set of questions, each of which typically had an embedded diagram, and then ask students to identify its parts or to complete elements that were missing. The diagram appeared on a whiteboard, and students used the whiteboard to answer the questions. This worked well.

2. A second lesson learned was the need to announce repeatedly that course reading resources were accessible to students. Many students were not doing the reading! Some were unaware that my lecture notes on JSP and JSD; summary articles by Jackson on JSP, JSD; and the book, *Problem Frames,* were all available in both Russian and English, and that they could be downloaded from the server and read at their convenience.

3. A third lesson has become quite obvious: The material in Problem Frames cannot be covered in a single semester, especially if the course includes JSP and JSD in addition to Problem Frames. More importantly, much of the reading in Problem Frames requires more background than 2nd year students have. Thus, I recommend that the 1st course, cover JSP, JSD and an Introduction to Problem Frames (Chapters 1–5; and part of Chapters 7 and 9, if time permits). This is what was covered in the first three iterations of the course. A 2nd course could be developed, 70 % of which would be devoted to *Problem Frames*, Chapters 6–11, after a brief review of JSP and JSD and Chapters 1–5 of *Problem Frames* (4 weeks).

Discussion: One might consider omitting JSP and JSD. Here are the reasons to include them:

a. D*esign* is about structure! Students learn about program design using JSP by developing a program's structure at the get-go; design and structure are par-

amount in software engineering. Students learn that *structure is key to intelligibility*;

b. Sequential data streams, the primary abstraction in JSP, are ubiquitous in software design;
c. the class of simple programs to which JSP applies is one of the 5 elementary Problem Frames;
d. finally, Jackson he is an exceptional writer and broadly-educated thinker. His ideas evolve as he increase his scope from programs (JSP) to real-time information systems (JSD) to complex cyber-physical systems (Problem Frames Approach)

4. Students meed more practice creating program and entity structure diagrams. Students practice creating simple structure (tree) diagrams at the outset. But many do not fully grasp the idea – additional exercises are thus needed.

5. Students need help in creating a list of operations needed for a program. Allocating each operation properly to the basic program structure is one of the steps in JSP Although JSP is language independent, I have used Java[17]. a language unfamiliar to most 2nd-year students. To make life easier for students, I started providing the Java operations needed by a program. I need to do this systematically.

6. Provide more practice reading and writing clear development descriptions. Problem Frames is an intellectual approach to structured analysis of problems. Analysis consists of writing three development descriptions in clear, concise natural language (Russian or English).

7. Teach and practice the syntax and semantics of state diagrams. Natural language descriptions can be augmented by state diagrams to give more precision. We can learn what a good state diagram looks like by studying the state diagrams created by Jackson in Problem Frames.

3.3 Future Work

My goal is to continue contributing to the training of software engineers by introducing them to Jackson's ideas and methods, and especially by reading, understanding and practicing the intellectual approach of Problem Frames.

During the coming year, I hope to make the following modest progress:

1. Significantly expand the bi-lingual Question Bank questions dealing with Problem Frames from 20 to 100.

2. Improve the Russian translation of the bi-lingual database of the course Question Bank as a whole.

3. Develop a database of student project work

4. Arrange for the publication of Problem Frames[18].

---

[17] I only have the code generator for Java which is by the CASE tool to generate programs from the elaborated structure diagram; in previous years, I have used code generators for Pascal, BASIC and C for early version of Windows.

[18] *Problem Frames* was translated under my direction during 2006-2007 during and immediately following my Fulbright Senior Scholar appointment at Petrozavodsk State University (PSU) during the 2005-2006 academic year by Irina Ossipian. We held numerous discussions with the Rectorate of PSU about publication by PSU; I contacted the publisher to arrange an (affordable) contract giving PSU permission to publish; but PSU then elected not to publish. I plan to undertake a review of the translation using translator resources at USTU; to include a set of simple corrections to the orig-

5. Disseminate the curriculum to several universities in Russia and elsewhere.

List of Sources

1. Bergland. “Structured Design Methodologies”. Annual ACM IEEE Design Automation Conference, Proceedings of the no 15 design automation conference on Design automation, Las Vegas, Nevada, United States, June 19–21, 1978.

2. Cameron, John. JSP&JSD: The Jackson Approach to Software Development. IEEE Computer Society Press. 1983, 1989.

3. Jackson M. A. Problem Frames: Analyzing and Structuring Software Development Problems, Addison-Wesley, 2001.

4. Jackson M. A. Software Requirements & Specifications, Addison-Wesley and ACM Press, 1996.

5. Jackson M. A. System Development, Prentice-Hall, 1983.

6. Jackson M. A. Principles of Program Design, Academic Press, 1975.

7. Jackson M. A. JSP in Perspective; in Software Pioneers: Contributions to Software Engineering; Manfred Broy, Ernst Denert eds; Springer, 2002.

8. Jackson M. A. Getting It Wrong: A Cautionary Tale; an oral contribution to program design courses; reprinted in JSP & JSD: The Jackson Approach to Software Development; John Cameron ed; IEEE CS Press, 1989.

9. Jackson M. A. A System Development Method; in Tools and Notions for Program Construction, Cambridge University Press, 1982, pp. 1–26.

10. Jackson M. A. Problem Analysis and Structure; in Engineering Theories of Software Construction, Tony Hoare, Manfred Broy and Ralf Steinbruggen eds: Proceedings of NATO Summer School, Marktoberdorf, IOS Press, 2000, pp. 3–20.

11. Jackson M. A. A Behaviour Manifesto for Cyber-Physical Systems, Draft chapter of forthcoming publication prepared for Proceedings of LASER Summer School, Elba, September 7–14, 2014. (The forthcoming publication has the title, Behaviours as Design Components of Cyber-Physical Systems).

12. Jackson M. A. Seven Truths of formal methods Draft of 10th June 2013.

13. Jackson M. A. Formalism and Intuition in Software Engineering;s in Juergen Muench and Klaus Schmid eds, Perspectives on the Future of Software Engineering: a Festschrift in Honour of Dieter Rombach, Springer verlag, 2013.

14. Jackson M. A. Topsy-Turvy Requirements; in Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday, Norbert Seyff and Anne Koziolek eds, Verlagshaus Monsenstein und Vannerdat, Muenster, 2012.

15. Jackson M. A. What Can We Expect From Program Verification? IEEE Computer, 2006, vol. 39, no. 10, pp. 53–59.

16. Jackson M. A. The Structure of Software Development Thought; in Structure for Dependability: Computer-Based Systems from an Interdisciplinary

inal machine-readable manuscript text, both of which were provided to me by Jackson in 2006; and to make correction to figures, many of which need to use a reduced font size to make all of the annotations visible.

Perspective, Besnard D, Gacek C and Jones CB eds, Springer, 2006, pp. 228–253, ISBN 1-84628-110-5.

17. Jackson M. A. A Discipline of Description; Proceedings of CEIRE98, Special Issue of Requirements Engineering, vol. 3 no. 2, pp. 73–78, 1998.

18. Jackson M. A. Object Orientation: Classification Considered Harmful; Proceedings of NordDATA'91, Oslo, 16–19 June, 1991, pp. 107–121.

19. Jackson M. A. The Origins of JSP and JSD: a Personal Recollection; IEEE Annals of Software Engineering, 2000, vol. 22, no. 2, pp. 61–63, 66.

20. Jackson M. A. The Boating Pond; an oral contribution to system development courses; published as Section 1.6 of: Michael Jackson; System Development; Prentice-Hall International, 1983.

21. Jackson M. A., Cotterman W. W., Couger J. D., Enger N. L., Harold F. eds Some Principles Underlying a System Development Method; in Systems Analysis and Design: a Foundation for the 1980's, North-Holland, 1981, pp. 185–194.

22. Information Systems: Modeling, Sequencing and Transformations; Proceedings of the 3rd International Conference On Software Engineering, pp. 72–81; IEEE 1979; reprinted in R. M. McKeag and A. M. McNaughten eds; On the Construction of Programs; Cambridge University Press, 1980, pp. 319–341.

23. Constructive Methods of Program Design; Proceedings of the 1st Conference of the European Cooperation in Informatics; G Goos & J Hartmanis eds; Springer-Verlag LNCS 44, 1976, pp. 236–262.

24. (with Pamela Zave) Pamela Zave and Michael Jackson; Four Dark Corners of Requirements Engineering; ACM Transactions on Software Engineering and Methodology, 1996, vol. 6 no. 1 pp. 1–30.

25. Ourusoff N. Reinvigorating the Software Engineering Curriculum with Jackson's Methods and Ideas, ACM SIGCSE Quarterly Bulletin, June, 2004.

26. Ourusoff N. An Introduction to Software Engineering: Jackson Structured Programming (JSP) and a little Jackson System Development (JSD) (manuscript developed from lectures, used in teaching Jackson's Software Engineering Methods during the period 1991–2007).

27. Van Wyk, Christopher J. "Processing Transactions" in Literate Programming, Communications of the ACM, vol. 30, no. 12, pp. 1000–1010.