Using Jackson Structured Programming (JSP) and Jackson Workbench to Teach Program Design

Nicholas Ourusoff
University of Maine at Augusta, Augusta, Maine, USA

nourusoff@yahoo.com

Abstract

Teaching how to program independently of teaching a programming language has been recognized as a worthwhile goal in computer science pedagogy, but many have abandoned the goal as being impossible to achieve in practice.

Jackson Structured Programming (JSP) is a well-documented and proven program design method that is independent of any programming language.

CASE tools have generally been used in designing information systems rather than programs. Jackson Workbench (Keyword Computer Services Limited, 2002) is a CASE tool for designing programs (as well as information systems) that generates executable program code in several contemporary programming languages (Visual BASIC, Java, C++). Jackson Workbench contains a unique Structure Editor that uses "hotspots" to draw and syntactically validate program tree structure diagrams. As a result, the user can focus entirely on the design process, and leave the details of drawing to the CASE tool.

Keywords: program design, visual design, design patterns, software engineering, constructive design, JSP, tree diagrams, modeling, computer science education, CASE tool

Introduction

Teaching how to program independently of teaching a programming language has been recognized as a worthwhile goal in pedagogy (See, for example Shackelford, 1998), but although desirable, many have abandoned the goal as being impossible to achieve in practice. Relying entirely on pseudo-code, for example, has significant drawbacks: students lose interest if they don't see a program run, and unless one has a tool to translate pseudo-code into executable code, the resulting paper designs are error-prone and boring to students.

Thus, most computer science programs have abandoned a language-independent approach to teaching programming. Yet, we all know that many teachers lament the fact that too much time in an introductory course in programming is spent learning the syntax of a programming language and how to debug syntax errors, both of which detract from the goal of teaching how to design.

What other program design methods are available? Historically, modular programming, which showed that good design incorporated modules, each of which implemented a specific goal, has been recognized since the beginning of higher-level programming (Glass, 1997). Structured programming showed that any program or module could and should be designed using well-formed constructs for sequence, selection and iteration (Wirth, 1969; Dijkstra, 1976). However, neither modular programming nor structured programming guarantees that a program will be well designed (Jackson,

1976). While they contain design principles, they are not a design method. A design method should provide a roadmap at each stage for choosing the best among competing design alternatives and offer a means of validating design decisions as they are made (Jackson, 1976).

Top-down structured programming and stepwise iterative design are two other program design approaches introduced in the 1970s. The main criticism of top-down structured programming is that it attempts to design in the vacuum that exists at the beginning of studying any problem. How can one make design decisions before one has a detailed idea of the problem? (Jackson, 1976, 1991)

Stepwise refinement was offered as a method of designing programs (Wirth, 1971). However, one can say that stepwise refinement – like top-down design – does not provide decision criteria for deciding what design choices should be made at each stage of the design process. Thus, stepwise design is an approach, but not a method.

Much contemporary work in programming is focused on formal methods of program derivation and verification (See, for example, (Hoare, 1969; Dijkstra, 1976; Gries, 1981). In *Constructive Methods of Program Design* (Jackson, 1976b), Jackson mentions that one can formally derive a correct program structure using JSP from the grammars that describe the input(s) and output(s). Hoare and Sridhar (1989) provide a formal proof of the equivalence of Jackson System Development (JSD) to CSP. Nevertheless, JSP is usually taught as an informal design method that takes advantage of various *program patterns* to motivate correct program structure.

The strongest arguments for the JSP and JSD design methods is that, in addition to their *soundness*, they are *practical* and *teachable* software engineering methods that lend themselves to visual CASE software engineering tools.

I have used Jackson Workbench to teach program design using JSP twice, in an introductory course in programming at the University of Maine at Augusta; and in a graduate-level course to 1st year Master's students at Petrozavodsk State University in Russia. In both courses, Jackson Workbench was also used introduce object-based information systems design using JSD about two-thirds of the way through the course. Students were given the strong impression that software engineering using a CASE tool was appropriate in designing both programs and information systems.

I found in both cases that students gained an appreciation for what designing programs with a correct structure entails; and that Jackson Workbench provided a very positive environment for designing and implementing working Java programs. Some students had a hard time learning to produce correct data models for inputs and outputs, learning to construct an elaborated program structure, and learning to construct and allocate operations to the program structure carefully and in detail. But most did learn to use JSP to design simple programs; and Jackson Workbench certainly made the task of producing tree diagrams and generating Java code enjoyable and quite effortless.

Organization

Section 2 contains a brief explanation of JSP. Section 3 introduces the Jackson Workbench CASE tool. Section 4 is devoted to illustrating the use of the Jackson Workbench to design a simple program. Section 5 concludes with some suggestions and challenges for further pedagogical work.

Jackson Structured Programming (JSP)

Jackson Structured Programming was developed in the 1970's by Michael Jackson (1976), and became a widely used design method, especially in Europe. At the World Health Organization, for example, JSP was used in the late 1970's and 1980's as a standard for specifying programs, while JSP was a government-wide specification standard in the UK,

Jackson Structure Diagrams (Tree Diagrams)

In JSP, design is defined as the relationship between parts to the whole. Programs have two types of components: composite components, which have a structure; and elementary components, which are elementary. Examples of elementary components are assembly language instructions or simple statements of a programming language. The composite components -- components having one or more parts -- are the three control structures of structured programming, sequence, selection and iteration. They are shown below, together with their textual representation in Jackson structure text, a pseudo code invented by Jackson, and ordinary pseudo code:

(a) sequence - a sequence is a composite component that has two or more parts occurring once each, in order (Figure 1).

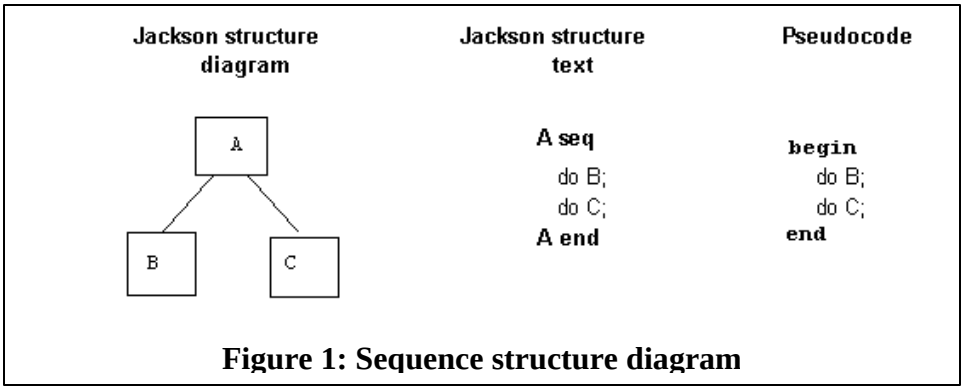

Figure 1: Sequence structure diagram

(b) selection - a selection is a composite component that consists of two or more parts, only one of which is selected, once. (Figure 2)

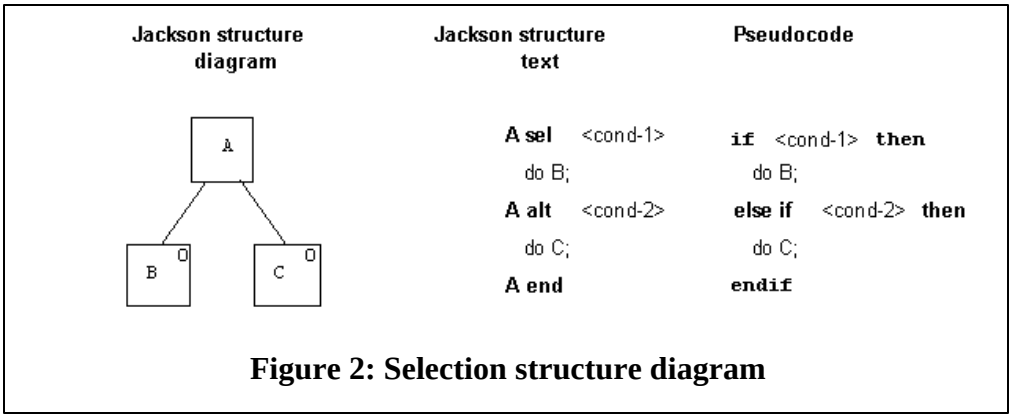

Figure 2: Selection structure diagram

(c) iteration - an iteration is a composite component that consists of one part that repeats zero or more times. (Figure 3)

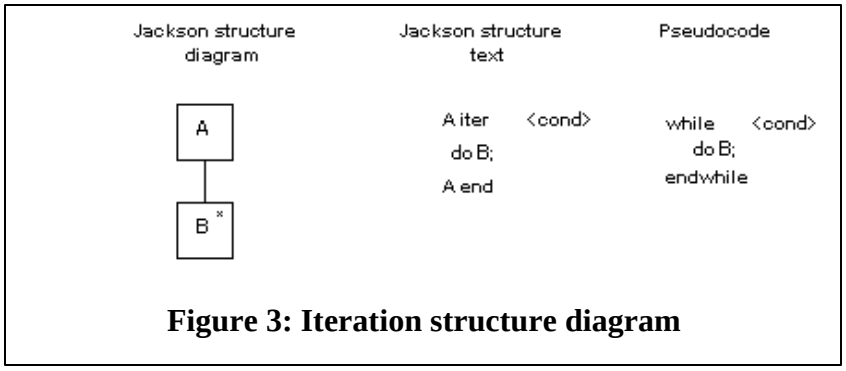

Figure 3: Iteration structure diagram

Basic JSP Design Method

JSP consists of the following steps:

1. Draw system diagram. (This step may be omitted when obvious.)
2. Draw data structures for program input(s) and output(s).
3. Form program structure based on the data structures from the previous step.
4. List and allocate operations to the program structure.
5. Create the elaborated program structure with operations and conditions added to the basic program structure
6. Translate the structure diagram into structure text or program code.

The result of applying JSP is a program that *reflects problem structure* as expressed in a model of its inputs and outputs. If changes to the program are required that only affect local components, the changes can be easily made to corresponding program components. A program's structural integrity – its correspondence with the problem's structure – is the primary way that we can reduce errors and costs in software maintenance.

In the next section, the basic JSP design method will be illustrated with a concrete example using Jackson Workbench.

Program Design Patterns

Jackson introduces a number of rules or programming *patterns* that were derived from observation and experience in processing sequential data streams: single read-ahead rule, group-id rule, multiple read-ahead rule, backtracking, structure clashes and program decomposition, and program inversion. These patterns, which are similar to the design patterns encountered in object-oriented design (OOD), are encountered in elementary programming and reflected in the designs produced using Jackson Workbench. Recognizing program design patterns while learning basic program design prepares students for later OOD design patterns encountered in designing information systems.

Using Jackson Workbench to Teach JSP

Jackson Workbench is a CASE tool for implementing Jackson design methods. It contains an intuitive JSP Structure Editor for implementing the Jackson Structured Programming (JSP) method and a JSD Network Editor for implementing the Jackson System Development (JSD) method. In this paper I only illustrate JSP using the Jackson Structure Editor.

The JSP Structure Editor is well documented in the Help provided as well as in examples that show how to generate Java code using the overall header, header, and trailer features.

One of the main strengths of the Structure Editor is its "smart" editing feature. The user never "draws: most tasks can be achieved with a single click thanks largely to a number of "hotspot" images that appear at different locations on the structure diagram indicating an operation that can be performed at that point simply by left-clicking. I have not found a more intuitive, easy-to-use structure editor. Use of Jackson Workbench to design the following simple program is illustrated in the following pages:

Multiplication Problem Statement: The basic design method can be illustrated by the following example: The lower-left triangle of a multiplication table is to be generated and printed. The required output is:

1									
2	4								
3	6	9							
4	8	12	16						
5	10	15	20	25					
6	12	18	24	30	36				
7	14	21	28	35	42	49			
8	16	24	32	40	48	56	64		
9	18	27	36	45	54	63	72	81	
10	20	30	40	50	60	70	80	90	100

Step 1 Draw system diagram. (This step is omitted since it is obvious.)

Step 2 Draw data structures for program input(s) and output(s). In our example, we only have the output to consider. (Figure 4

First, we open a new JSP file and name it (Figure 4).

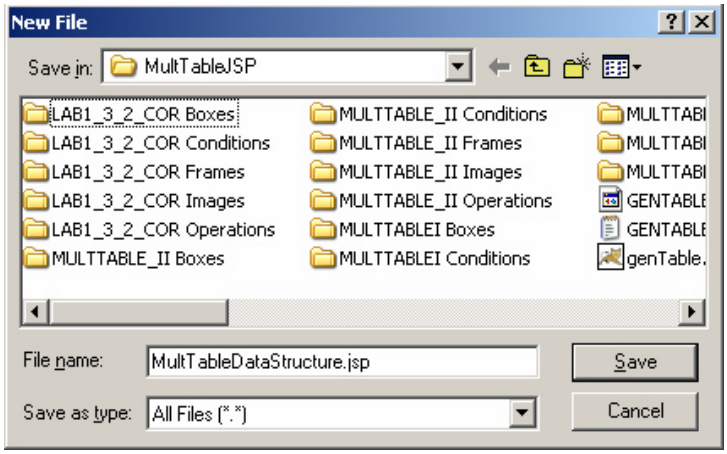

Figure 4: Opening a newfile

The JSP Structure Editor produces the initial structure shown in Figure 5:

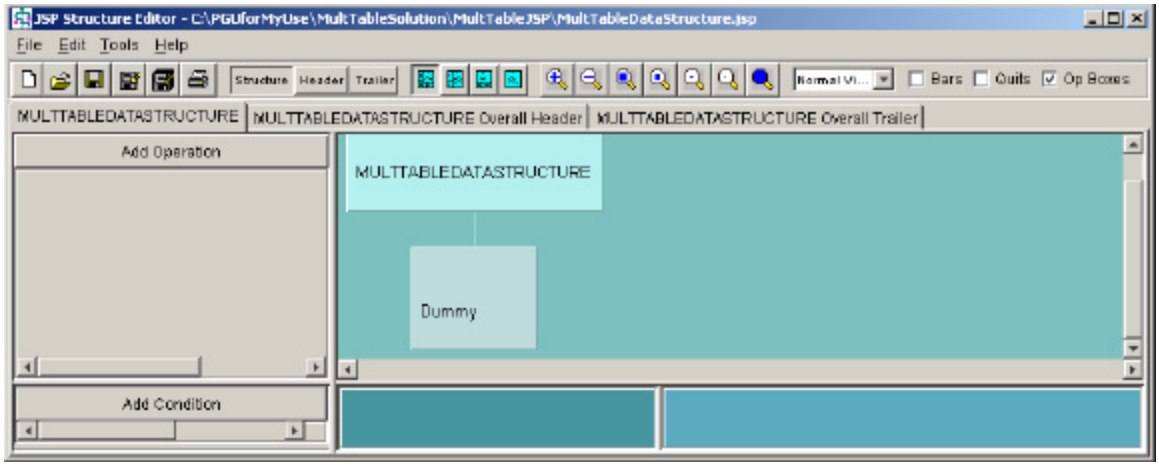

Figure 5: Initial structure produced by Jackson Workbench

Double-clicking on the box labeled Dummy causes the label to be surrounded by a white rectangle, indicating that the text can be selected and over-written (Figure 6):

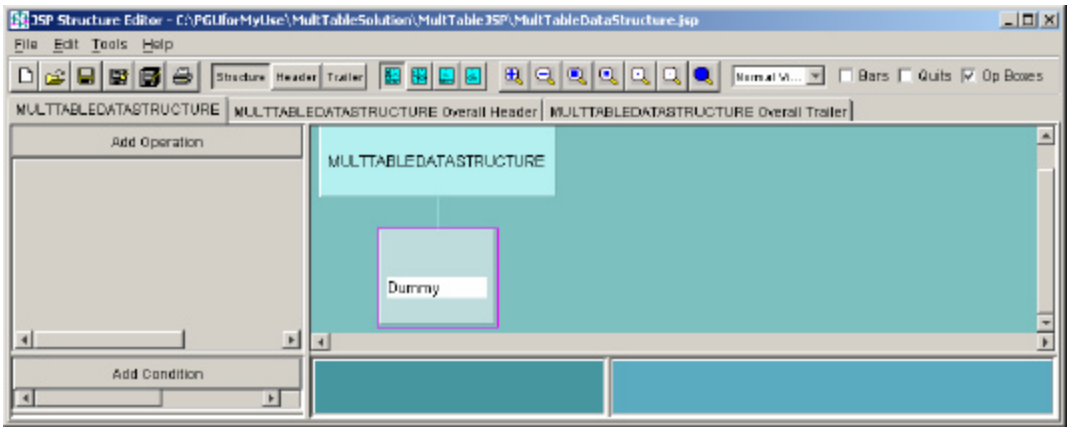

Figure 6: Replacing "Dummy"

Replacing Dummy with row as the label, and preparing to change type to iteration: dragging the mouse to the upper right-hand corner causes "Change box type" to appear (Figure 7).

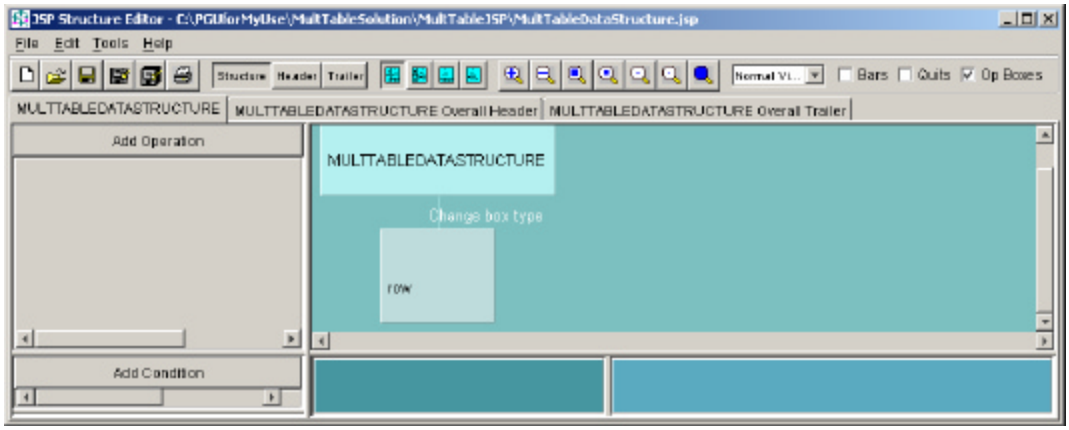

Figure 7: Changing Box type

Clicking the mouse will cause the symbol for an iterated component to appear in the upper-right corner of the box (Figure 8):

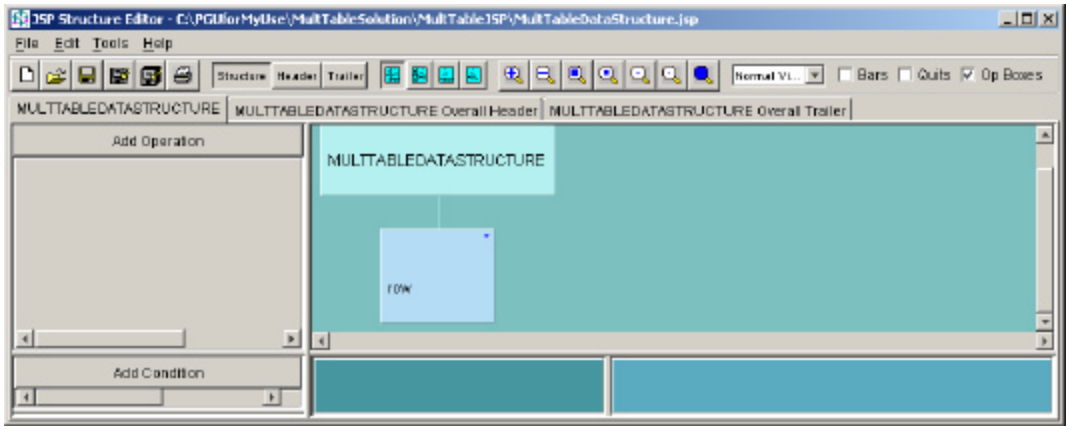

Figure 8: Box Type changed to an iterated component

Moving the mouse to bottom of box labeled “row” in order to add a sub-tree causes a "hotspot" image to appear just below, indicating that a new sub-tree can be added by positioning the mouse over the “hot-spot" and left-clicking (Figure 9):

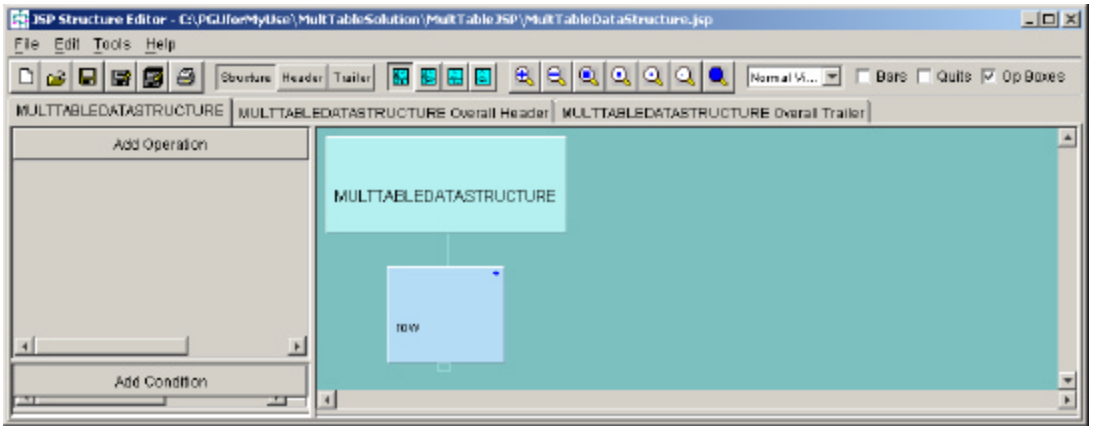

Figure 9: “hotspot” image showing placeholder for new sub-tree

A new sub-tree is added after clicking the small rectangle centered below the iterated row component row (Figure 10):

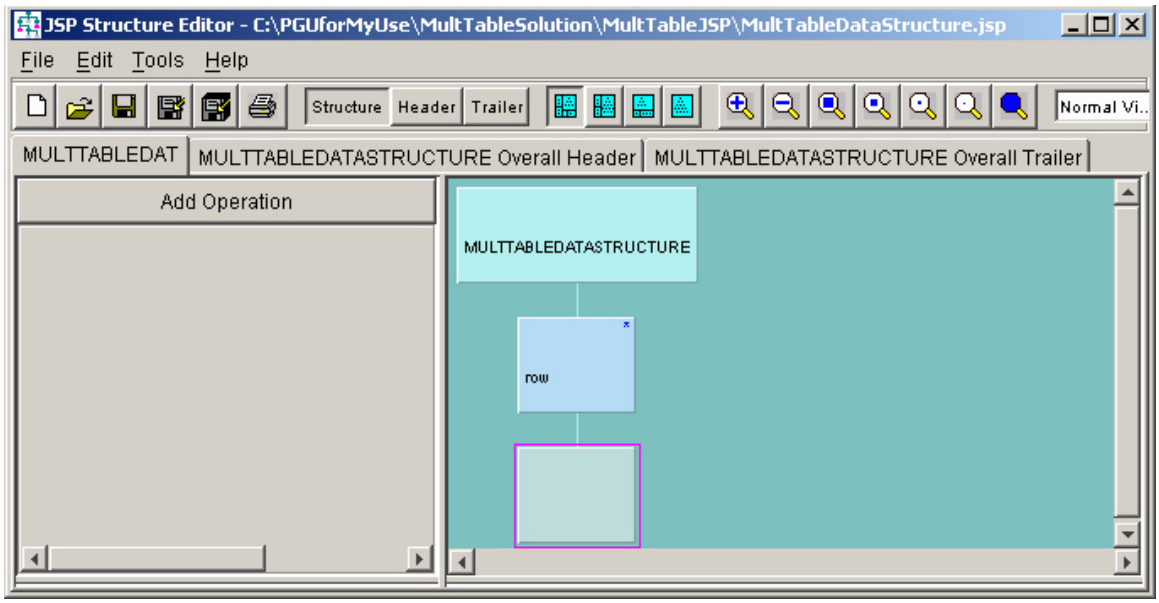

Figure 10: A ne w sub-tree added

In Figure 11 below, we complete the data structure for multiplication table by labeling the new sub-tree with “element” and changing the box type to an iterated component:

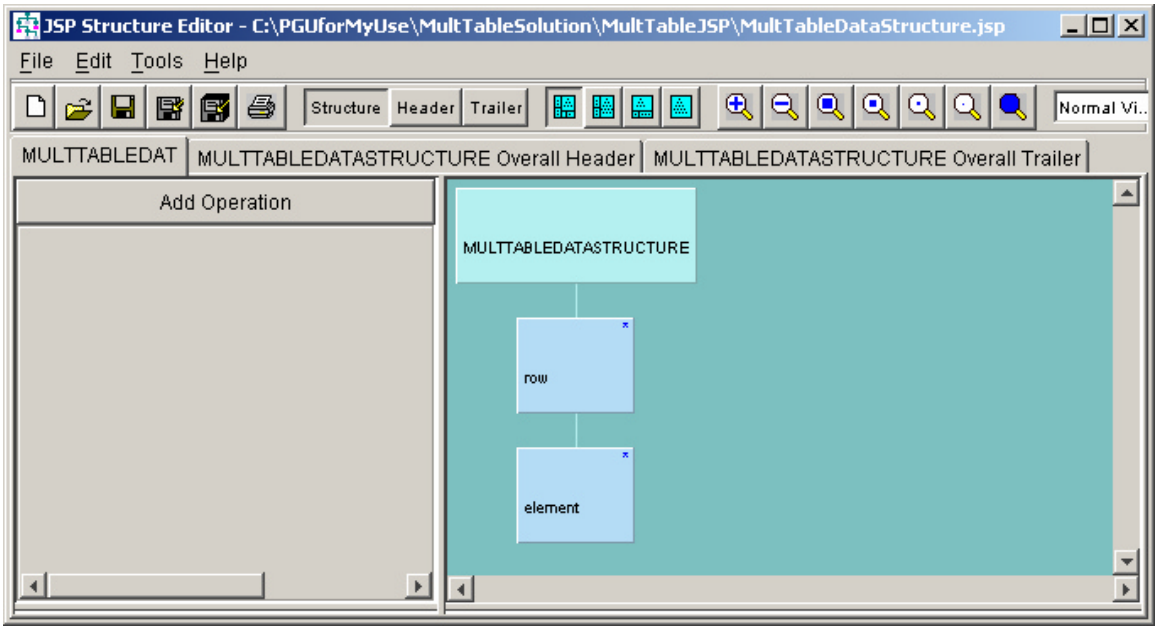

Figure 11: Completed data structure for multiplication table

Step 3 Form program structure based on the data structures from the previous step by adding verbs to each component to make them actions

This is shown in Figure 12 below:

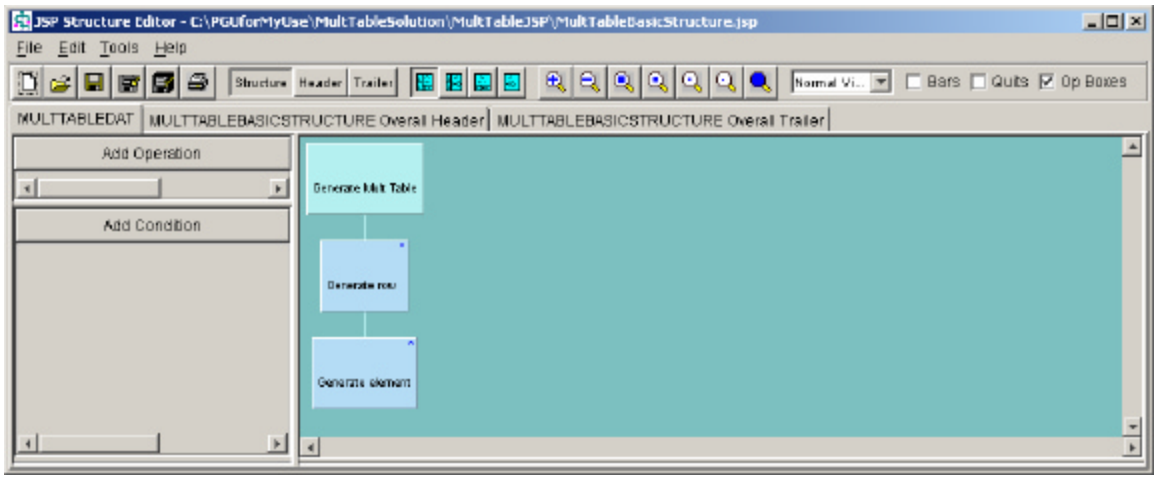

Figure 12: Basic program structure

Step 4 List and allocate operations

We list the elementary operations needed to perform the task, and answer for each operation, "How often is it executed?" and "In what program component(s) does it belong?" The operations must be elementary statements of some programming language; we have chosen Java.

#	Operation	how often?	where?
1	MAX=10;	Once	at start
2	row=1;	Once	at start
3	col=1;	once per row	in component that produces a table row, at start of component
4	row++;	9 times	in component that produces a table row, at end of component
5	col++;	55 times	in component that produces a table element, at end of component
6	element = row*col;	once per element	in component that produces a table element
7	System.out.print(element);	55 times	in component that produces a table element

Note: Although Jackson Workbench is not used in this step, we include the table of operations in order to make subsequent steps intelligible.

Step 5. Create the elaborated program structure with operations and conditions added to the basic program structure

After having created a table containing a list of operations needed together with the number of times each occurs and in what component each appears, we are ready to create the elaborated program structure from the basic program structure. We click the Bars on the menu bar, and click Add Operation.

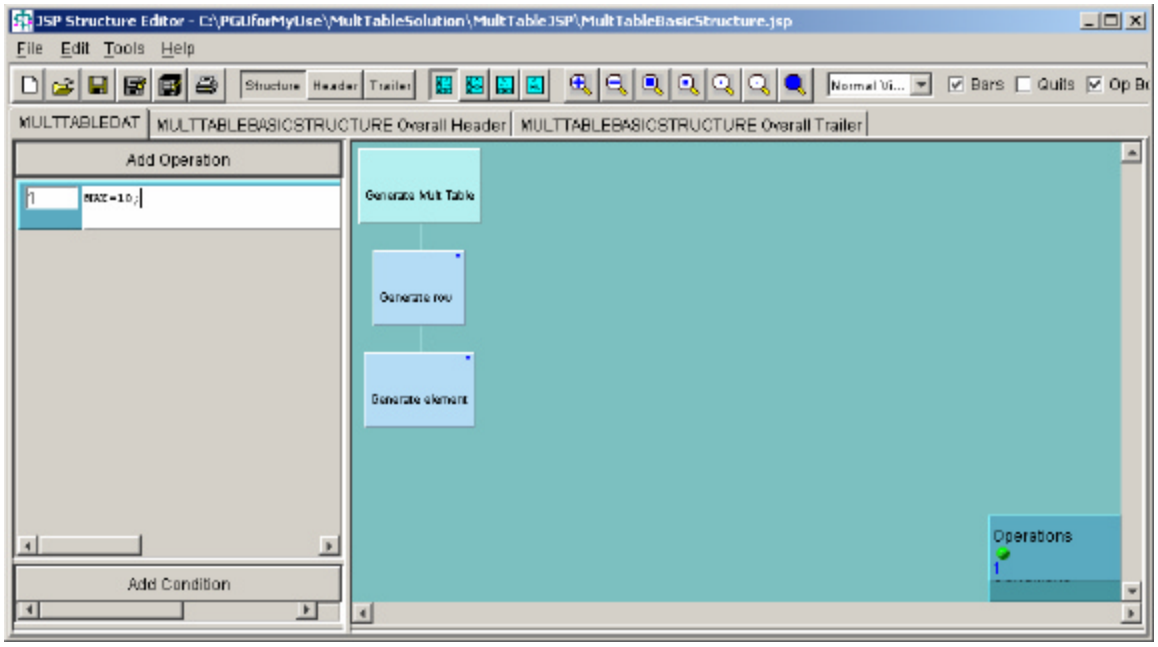

Figure 13: Adding an operation

The Operations status box (lower right) adds the 1st operation, and a space opens up under Add Operations to add the 1st operation. We type MAX=10 (Figure 13)

At this point, we must insert a new level into the structure diagram. We wish to allocate the 1st operation at the start as the leftmost leaf of the root node. But, since the root is an iteration, we must convert it first into a sequence consisting of our 1st operation and a component, Generate rows, an iteration with one component, Generate row. (Otherwise, we would have under the root our 1st operation followed by a component that repeats – and the root would be neither a sequence nor an iteration - it would be a badly-formed component. We first position the mouse just under the root, and when a small box appears, click on it to create a new box under the root. We then slide the mouse on the arc connecting the root to the new box, and move it left: a small box will appear – it is shown below (just beneath Generate Mult Table), and, if we click on it, we create the box as a left sibling of the new box under the root (Figure 14):

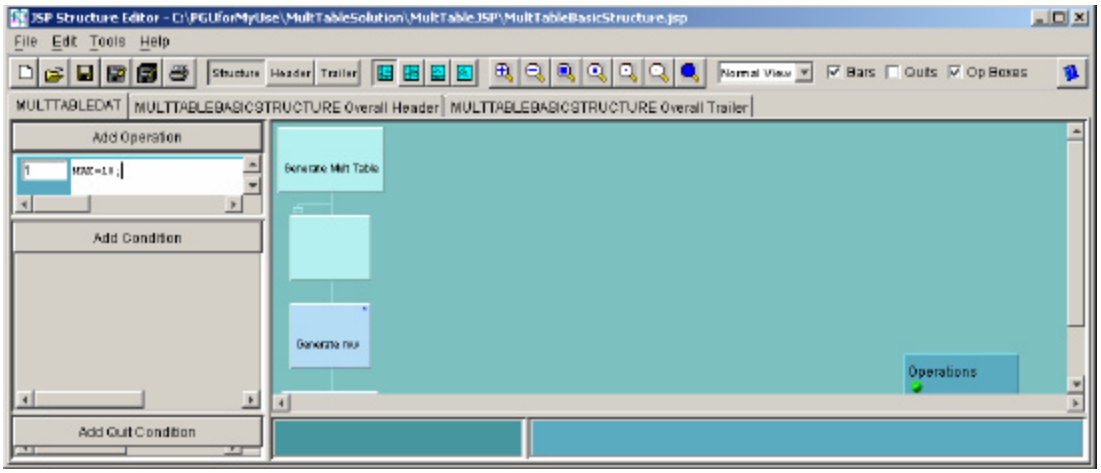

Figure 14: Inserting an intermediate level

After we click on the empty box shown above, we allocate the 1st operation to the leftmost box, by dragging the 1 from the Operation box (lower right) into the leftmost leaf of our root. Then, we add a label to the new box to its right (Figure 15):

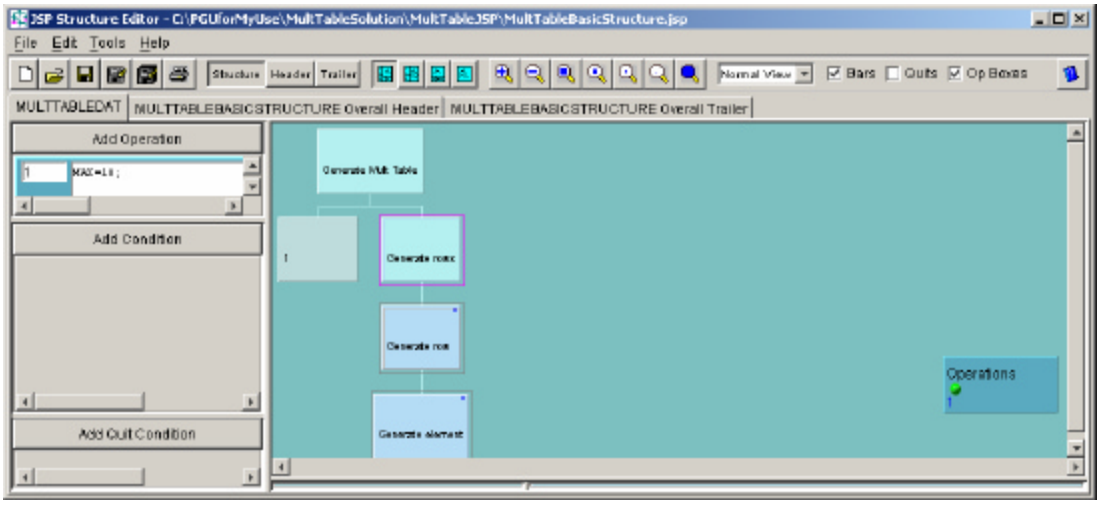

Figure 15: Allocate an operation to new level

Next, we will add a condition to our program structure. The condition for continuing the iteration, Generate rows, is that row is less than equal to MAX (we model iteration using a while construction here). We click Add Condition, and a space opens up under the Add Condition menu bar on the left, and a 1 appears in the Condition bar (lower right). We add the condition, row <= MAX. Then, we drag the 1 to the Generate row box (Figure 16):

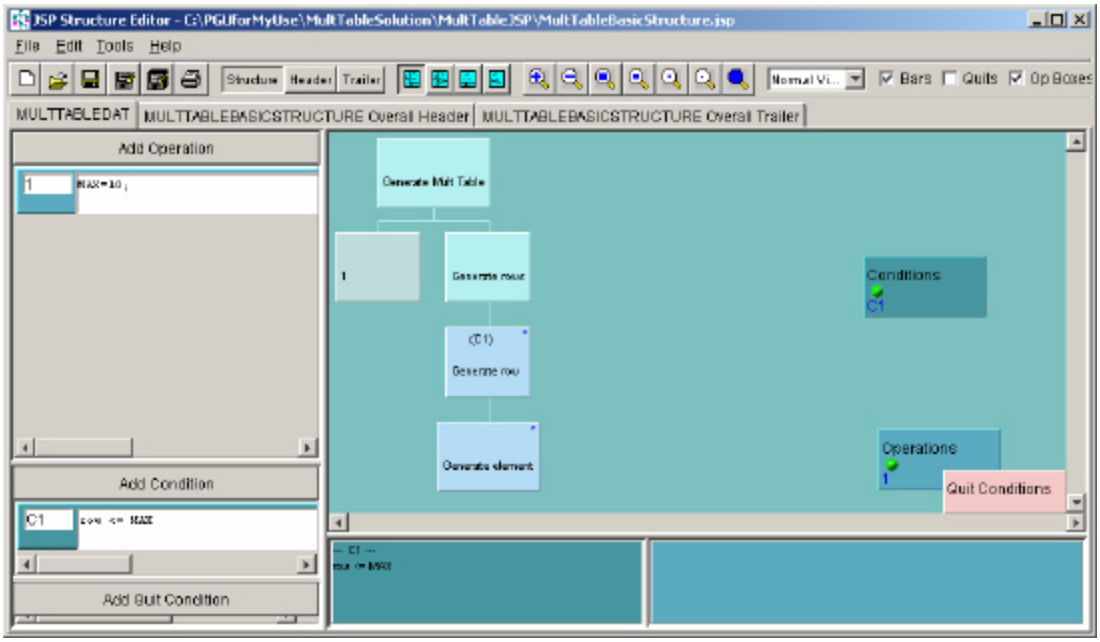

Figure 16: Adding a condition

We complete the elaborated program structure by adding other operations and conditions (Figure 17):

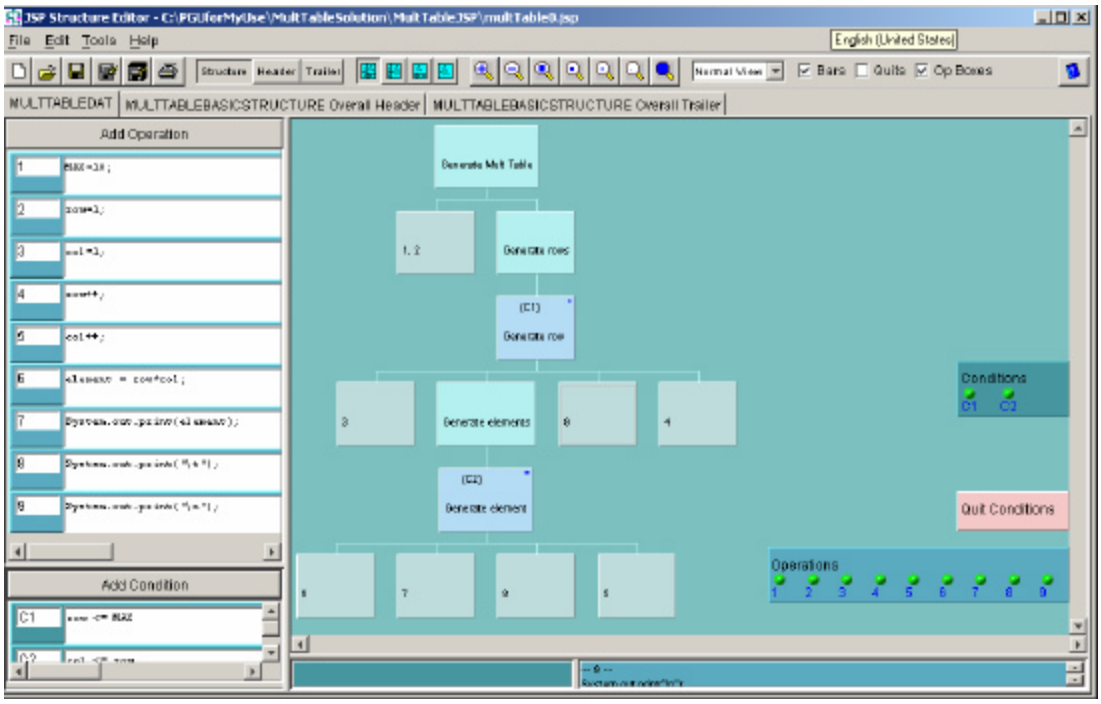

Figure 17: Elaborated program structure

To generate a Java program, we need to add the class header as the overall header, the header for main as the header together with primitive type declarations, and a trailer consisting of a right curly brace, }. The Java program together with its output generated from the elaborated program structure is shown below in Figures 18 and 19.

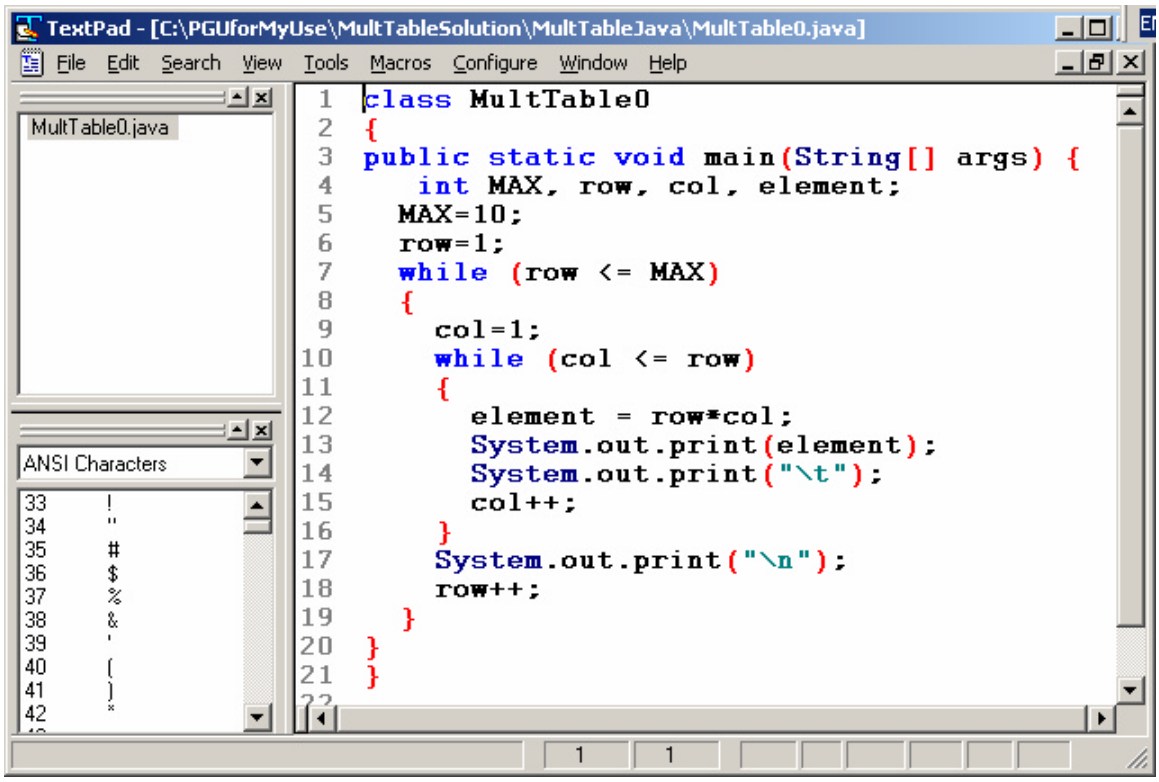

Figure 18: Generated Java code

Conclusion

The quality of contemporary software leaves much to be desired (See [Jackson, 2001], p.331 for an example in a flagship product with millions of copies sold.) Our best hope for producing better software is to enforce a software engineering discipline that focuses on problem analysis and design. In this paper, I have illustrated the use of Jackson Workbench and JSP to inculcate students from the outset with a soft-

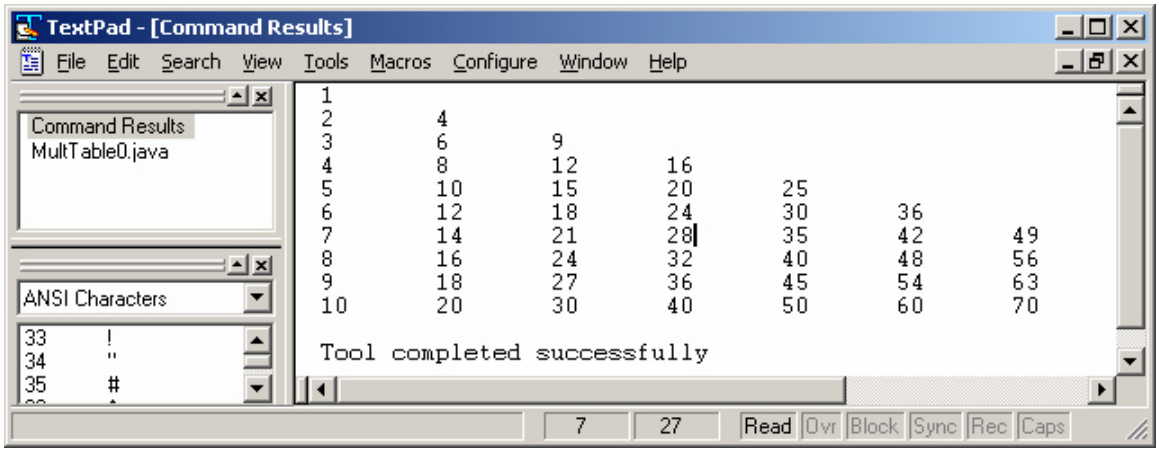

Figure 19: Output produced by generated Java program

ware engineering approach to designing programs.

A subsequent part of this pedagogy continues with the design of information systems using Jackson System Development (JSD) – it is object-based and uses the design techniques of JSP – in conjunction with the Network Editor of Jackson Workbench. I suggest that Jackson Workbench and Jackson methodology (JSP and JSD) could form a promising introductory software engineering curriculum.

Further experience and evaluation of this approach is certainly needed. And, Jackson Workbench is not a finished product and its author does not have a timetable for completing it. Although Jackson Workbench is adequate to design and generate code for programs using JSP, I have only used it for very simple information system problems using JSD.

Finally, we should train students to think in terms of problem classification and decomposition. Just as we do not wish to jump into programming without proper design, we do not wish to jump into design without proper analysis.

References

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Glass, Robert L. (1997). *In the Beginning: Recollections of Software Pioneers*. Wiley-IEEE Press.

Gries, David. (1981). *The Science of Programming*. Springer-Verlag.

Hoare, C. A. R. (1969). An Axiomatic basis for computer programming. *Communications of the ACM*. *12*(10): 576-580.

Hoare, C.A.R. & Sridhar, K.T. (1985). *JSD Expressed in CSP in Data Types and Persistence*. (Appin). Informal Proceedings 1985: 49-82. (Reprinted in "JSP&JSD: The Jackson Approach to Software Methodology" by John Cameron. IEEE Computer Society Press (2nd edition), 1989).

Jackson, Michael. (2001). *Problem Frames*. Addison-Wesley.

Jackson, M. (1995). *Software Requirements & Specifications: A Lexicon of practice, Principles and Prejudices*. ACM Press/Addison-Wesley.

Jackson, M. (1983). *System development*. Prentice-Hall International.

Jackson, M. (1976a). *Principles of Program Design*. Academic Press.

Jackson, M. (1976b). *Constructive Methods of Program Design*. In Lecture Notes in Computer Science Vo lume 44, 1976, pages 236-262. Copyright 1976 by Springler-Verlag (See http://dspace.dial.pipex.com/jacksonma/ for a bibliography of Jackson's work.)

Keyword Computer Services Limited. Jackson Workbench. (2002). Contact: KCSL@BTInterent.comor Jim.Newport@BTInternet.com or +44 1494 870427). Version 2.04 has been used throughout.

Shackelford, Russel. (1998). *Introduction to Computing and Algorithms*. Addison-Wesley.

Wirth, Niklaus. (1971). Program Development by Stepwise Refinement. *Communications of the ACM*. 14(4): 221-227.

Biography

Nicholas Ourusoff learned to use JSP as a programmer-analyst for the World Health Organization from 1975-1979 and used JSP as a United Nations Census Data Processing expert in validating population and housing data for Senegal and Guine-Bissau. From 1984 until the present time, as a teacher of computer science and computer information systems, he has continued to follow the evolution of Jackson's thought and has tried to bring attention to Jackson's innovative work in program and information systems design and more recently in problem analysis.

Home Page: http://www.geocities.com/nourusoff