In M. Petre & D. Budgen (Eds) Proc. Joint Conf. EASE & PPIG 2003 Pages 347-357

First Results of An Experiment on Using Roles of Variables in Teaching

Marja Kuittinen and Jorma Sajaniemi
University of Joensuu
Department of Computer Science
P.O.Box 111, FIN-80101 Joensuu, Finland
{Marja.Kuittinen|Jorma.Sajaniemi}@Joensuu.Fi

Abstract

Roles of variables is a new concept that captures tacit expert knowledge in a form that can, e.g. be taught in introductory programming courses. A role describes some stereotypic use of variables, and only ten roles are needed to cover 99 % of all variables in novice-level programs.

This paper presents the first results of an experiment where roles were introduced to novices learning Pascal programming. Students were divided into three groups that were instructed differently: in the traditional way with no treatment of roles; using roles throughout the course; and using a role-based program animator in addition to using roles in teaching. The results suggest that the introduction of roles improves program comprehension and program writing skills. Moreover, the use of the animator affects the way students describe programs: they stress data-related issues concerning the deep structure of a program, as opposed to directly visible operations and control structures.

1. Introduction

Programming skills have been necessary ever since computers were invented. At the very beginning only few programmers were needed but as computers became more common the need for skilled programmers increased rapidly. At that time, programming was taught the best (and only) way known, "via syntax, through the vehicle of a single language" (Fincher, 1999). More recently, teaching methods that are considered to be good have been gathered and documented as collections of pedagogical patterns (Fincher and Utting, 2002). New efforts to ease and enhance learning have varied in their general approach to improve learning: most studies report effects of new teaching methods and new ways of presenting teaching materials, while reorganization of topics and introduction of new concepts have been far more rare.

Research into *teaching methods* covers studies exploring the usage of different ways to conduct teaching sessions, including lectures combined with discussion groups (e.g., Hagan *et al.*, 1998), problem solving (e.g., Davies, 1996; Feldman, 1999; Hanly and Koffman, 1999; Koffman, 1986), watching example code running or predicting what happens next (e.g., Pirolli and Anderson, 1985; Wiedenbeck, 1989), and learning by doing (e.g., Fleury, 1997; Jenkins,1998)). The other common approach, research into *forms of materials*, includes studies trying to explain the effect of presenting materials in different ways, such as the use of graphics and graphical metaphors in learning materials (e.g., McKay, 1999*a*, 1999*b*), and program and algorithm visualization and animation (see Hundhausen *et al.* (2002) for an overview). As an example of the third category, *reorganization of topics*, Ginat (2001) has studied possibilities to introduce algorithm efficiency considerations at an early phase of learning programming.

We know only two examples of the last category, research into *new concepts* that can be utilized in teaching elementary programming: software design patterns, and roles of variables. Software design patterns (Clancy and Linn, 1999) represent language and application independent solutions to commonly occurring design problems. The number of patterns is potentially unlimited, and there are sets of patterns for various levels of programming expertise (e.g., elementary patterns for novice programmers (Wallingford, 2003)) and application areas (e.g., data structures (Nguyen, 1998)). Research into the use of patterns indicates that instructors should expect to refine the patterns they offer students on a regular basis (Clancy and Linn, 1999).

Roles of variables (Sajaniemi, 2002, 2003) describe stereotypic usages of variables that occur in programs over and over again. Only ten roles are needed to cover 99 % of all variables in novice-level programming, and they can be described in a compact and easily understandable way (Sajaniemi, 2002). Ben-Ari and Sajaniemi (2003) have shown that in one hour's work, computer science teachers can learn roles and assign them successfully in normal cases. As op-

posed to the patterns approach, the set of roles is so small that it can be covered in full during an introductory programming course.

To find out the effects of using the role concept in teaching programming to novices, we conducted a teaching experiment with three experimental conditions: one group of students were instructed in the traditional way, another with roles covered during the course, and the third group with roles and role-based animation of programs. This paper reports the first results of the experiment.

The aim of teaching is to cause some change in a learner's knowledge and skills. There are a number of learning theories with different views on what changes should be favored and how these changes may be achieved (see, e.g., Hundhausen *et al.* (2002)). The result of learning can be generally characterized to be one of the following: a set of facts as they are described in the learning materials; a set of facts with effective access mechanisms (e.g., the dual-coding theory); a set of self-generated facts (e.g., the cognitive constructivism theory); a skill to apply given or self-generated facts in new situations; and finally, a full replication of experts' mental model (the epistemic fidelity theory).

Programming is a skill where knowledge about programming languages, programming techniques, and application domain are utilized to create new artifacts, i.e., new programs. Thus the purpose of teaching programming cannot be just an introduction of a set of facts but their application in new situations is also needed. On the other hand, in programming the differences between novices and experts are so huge that it is unreasonable to strive for epistemic fidelity in the first programming courses. Therefore, we set as our goal to give students programming knowledge and the skill to apply this knowledge in new situations, and we will measure our success on that level of learning.

The rest of this paper is organized as follows. Section 2 describes the role concept and its potential uses in teaching to program. Section 3 presents the experiment and discusses its results. Finally, Section 4 contains the conclusion.

2. Roles of Variables

Sajaniemi (2002) has introduced the concept of the *roles of variables* as a result of a search for a comprehensive, yet compact, set of characterizations of variables that can be used, e.g., for teaching programming and analysing large-scale programs. His work is based on earlier studies on variable use made by Ehrlich and Soloway (1984), Rist (1991), and Green and Cornah (1985). Roles are supposed to capture tacit expert knowledge – a view supported by the findings made by Ben-Ari and Sajaniemi (2003).

2.1 The Role Concept

A role describes the dynamic character of a variable embodied by the succession of values the variable obtains, and how the new values assigned to the variable relate to other variables. For example, in the role of a *stepper*, a variable is assigned a succession of values that is usually known in advance as soon as the succession starts – even though the length of the succession may be unknown. The role concept does not concern the way a variable is used in the program; only the succession of values, and their lifetimes, do matter.

```
program doubles (input, output);
var data, count, value: integer;
begin
    repeat
        write('Enter count: ');
        readln(data)
    until data > 0;
    count := data;
    while count > 0 do begin
        write('Enter value: ');
        readln(value);
        writeln('Two times ', value,
                ' is ', 2*value);
        count := count - 1
    end
end.
```

Figure 1. A short Pascal program.

As an example, consider the Pascal program in Figure 1. In the first loop, the user is requested to enter the number of values to be later processed in the second loop. The number, stored in the variable `data`, is requested repeatedly until a valid input is obtained. The variable `value` is used similarly in the second loop: there is no possibility for the programmer to guess what values the user will enter. Since these variables always hold the latest in a sequence of values, their role is said to be *most-recent holder*. The variable `count`, however, behaves very differently: once it has been initialized, its future values will be known exactly. It will step downwards one by one until it reaches its limiting value of zero. The role of this variable is that of a *stepper*.

Table 1 gives short descriptions of all roles; for a more comprehensive treatment, see the *Roles of Variables Home Page* (Sajaniemi, 2003). The *organizer* is the only special role for arrays; usually the role of an array is that of its elements, e.g. an array of *gatherers* is itself a *gatherer*.

The set of roles has been obtained through an analysis

Table 1. Informal role definitions.

Fixed value	A variable which is initialized without any calculation and whose value does not change thereafter.
Stepper	A variable stepping through a succession of values that can be predicted as soon as the succession starts.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values.
Most-wanted holder	A variable holding the "best" value encountered so far in going through a succession of values. There are no restrictions on how to measure the goodness of a value.
Gatherer	A variable accumulating the effect of individual values in going through a succession of values.
Transformation	A variable that always gets its new value from the same calculation from value(s) of other variable(s).
Follower	A variable that gets its values by following another variable.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Organizer	An array which is only used for rearranging its elements after initialization.
Temporary	A variable holding some value for a very short time only.
Other	Any other variable.

of all the programs in three elementary programming textbooks (Sajaniemi, 2002). In this analysis, the three most frequent roles, *fixed value*, *stepper* and *most-recent holder* accounted for 84% of all variables.

The role of a variable may change during the execution of a program and this happens usually somewhere between two loops. For example, in the program of Figure 1, the two variables `data` and `count` could be combined to a single variable, say `count` (making the assignment "`count := data;`" unnecessary). The role of this variable would first be a *most-recent holder* and then, in the second loop, a *stepper*.

It should be noted that roles are cognitive – rather than technical – concepts. As an example, consider the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ... where each number is the sum of the previous two numbers. A mathematician who knows the sequence well can probably see the sequence as clearly as anybody sees the sequence 1, 2, 3, 4, 5, ..., i.e., the continuum of natural numbers. On the other hand, for a novice who has never heard of the Fibonacci sequence before and who has just learned the way to compute it, each new number in this sequence is a surprise. Hence, the mathematician may consider a variable as stepping through a known succession of values (i.e., a *stepper*) while the novice considers it as a *gatherer* accumulating the previous values to obtain the next one.

2.2 Using Roles in Teaching

The set of roles is so small that it can be fully covered in an introductory programming course. As roles are tools for programming, they should not be taught as a separate issue but introduced gradually as they appear in programs. Even though there is a exact technical definition for each role, informal definitions (in the style of Table 1) are sufficient for novices.

In addition to schema knowledge concerning the roles themselves, role utilization includes strategic knowledge about their use in programming. For a novice it may be difficult to start to write a program: new programming concepts form an overwhelming set of fragile knowledge that is hard to apply (Davies, 1993) and the decision of what knowledge to apply first is not easy. This problem can diminished by guiding novices to start a programming task by thinking about data requirements: what roles (and consequently variables) are needed to cover the input and output requirements of the programming assignment, and what code sequences are typical for these roles.

Role knowledge can be further advanced by role-based program visualization and animation. PlanAni (Sajaniemi and Kuittinen, 2003) is a role-based program animator that uses role images for visualizing variables and role-based animation for visualizing operations. A role image – a visualization used for all variables of the role – gives clues on how the successive values of the variable relate to each other and to other variables. For example, a *most-wanted holder* is depicted by two flowers of different colors: a bright one for the current value, i.e., the best found so far, and a gray one for the previous, i.e., the next best, value.

Figure 2 is a screen shot of the PlanAni user interface. The left pane shows the animated program with a color enhancement showing the current action. The upper part of

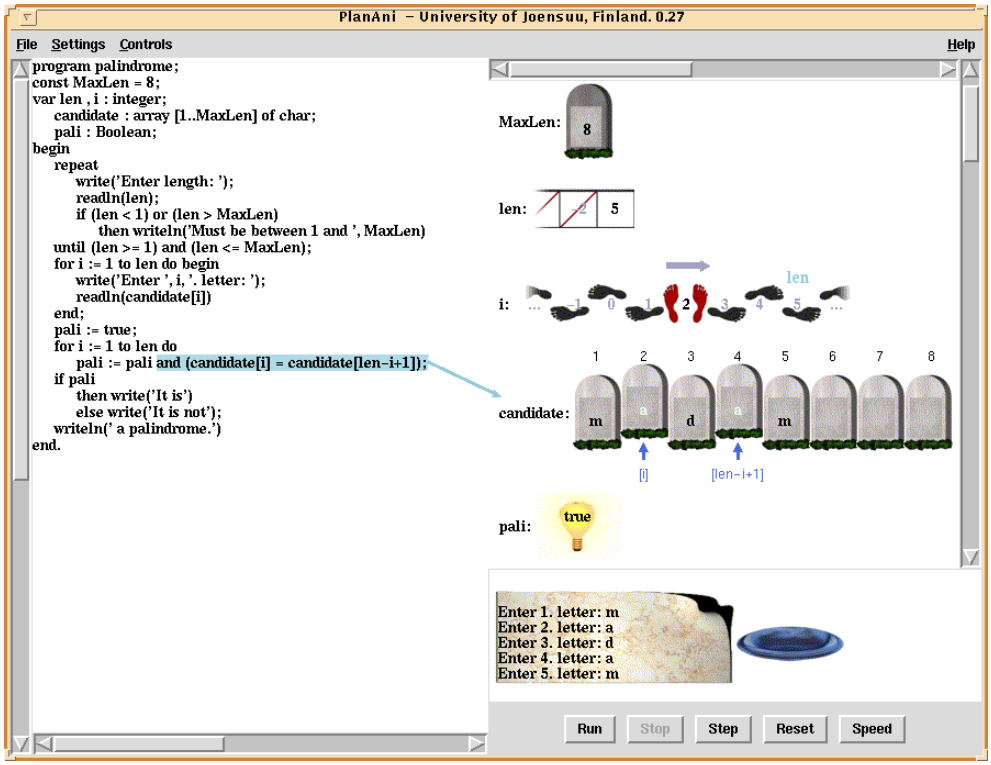

Figure 2. The user interface of the PlanAni program animator.

the right pane is reserved for the variables, and below it there is the input/output area consisting of a paper for output and a plate for input. The currently active action in the program pane on the left is connected with an arrow to the corresponding variables on the right. Whenever the color enhancement is moved to a new location in the program, the new enhancement flashes.

3. Experiment

To test the hypothesis that introducing roles of variables in teaching facilitates learning to program, we conducted an experiment during an introductory Pascal programming course at university level. Students were divided into three groups that were instructed differently: in the traditional way in which the course had been given several times before, i.e., with no specific treatment of roles; using roles throughout the course; and using a role-based program animator in exercises in addition to using roles in teaching. The course lasted five weeks, with four hours of lectures and two hours of exercises each week.

At the end of the course there was an examination which was graded normally for the purposes of the course. Students' answers were, however, analyzed for this experiment in other ways to find qualitative differences between the groups.

In order to prevent students from switching back and forth between groups, the lectures were scheduled to occur at the same time. As a consequence, two lecturers had to be used. Both teachers had a long experience in giving lectures to undergraduate students and both had taught the course before. The teacher giving traditional lectures did not know about the role concept. Thus, there were no negative effects of the teacher avoiding some issues in his lectures as he did not know what the experiment was exactly about. In order to find any differences caused by different teachers and students' different degrees of engagement in the course, the examination included questions that were not related to variables and thus were expected to yield similar results in all groups.

Both in the middle and at the end of the course, some students were given program comprehension and program creation tasks which were videotaped. These protocols will be analyzed later to find qualitative differences in the conceptual level of utterances students used when talking about programs.

3.1 Method

The experiment was a between-subject design with the content of instruction as the between-subject factor. The subjects were divided into three groups: one receiving

Table 2. Basic data about the experimental groups. In all scales higher values are better.

	Group				p
	Traditional	Roles	Animation	All	
Number of subjects	26	32	33	91	
Female subjects (%)	30.8	18.9	24.2	24.2	0.7380
High school mathematics average (scale 1-3)	2.4	2.2	2.1	2.2	0.1343
High school mother tongue average (scale 1-3)	2.4	2.2	2.1	2.2	0.1851
High school information technology average (scale 1-3)	2.5	2.6	2.6	2.6	0.9828
High school art average (scale 1-3)	2.2	2.3	2.4	2.3	0.7039
Spreadsheet usage average (scale 0-2)	1.3	1.0	1.2	1.1	0.2827
Programming courses average (scale 0-2)	0.8	0.9	1.0	0.9	0.9539
Programming experience average (scale 0-2)	0.8	0.5	0.8	0.7	0.4566

normal lectures and exercises (the *traditional group*), one attending lectures with systematic use of variables roles throughout the course (the *roles group*), and one attending the same lectures as the roles group but using role-based animator in exercises (the *animation group*).

All groups were presented the same instructional materials and example programs with the only exception being the presentation of roles. In the roles and animation groups, roles were introduced in the lectures gradually as they appeared in example programs. In the lecture hand-out the declaration of each variable had its role in a comment. Students were also given a printed list describing all roles (4 pages). In exercises, the role of each variable was mentioned to students. In the traditional group, the same amount of teaching time was spent without explicitly mentioning roles. The hand-outs were otherwise equivalent to the other groups except missing role names in variable comments. As a substitute to the role list, traditional group students were given the same programs as "further examples". During lectures, the same programs were explained to all groups.

During exercises, all groups executed four programs; one program in each exercise session except the first one. In these tasks, the animation group used role-based program animator PlanAni, and the other groups used a visual debugger (Turbo Pascal v. 7.0). Each exercise session started with students presenting their solutions to home assignments. Animations, lasting between 20 and 40 minutes, were always used at the end of the sessions. In each session, the teacher first presented the animation step by step using her computer and a video projector. In the animation group, the teacher explained for each new role what the role image was and how it tried to visualize the most important properties of the role. In all groups, students were then instructed to run the animation using given data, carefully selected by the teacher. Thereafter, students animated the program with their own input data. Finally, the teacher discussed with students about complicated issues or other problems students had in understanding the program. All the time, students were encouraged to proceed slowly with the animation and predict the effect of the next statement on the values of variables and other aspects of the program.

Hundhausen *et al.* (2002) argue that the way students use visualization technology has a greater impact on effectiveness than the content of the visualizations. By using the same tasks and activities in all groups, we have tried to make sure that the cognitive activities were in each group equivalent so that differences in their performance would not be due to differences in cognitive activities but to the content of the visualizations.

3.1.1 Subjects

The subjects were undergraduate students studying computer science for the first semester. Students attended the same first lecture where they filled out a short questionnaire which solicited information concerning their high school grades and their previous experience with computers and computer programming. After the first lecture, students (n=80) were randomly divided into three groups, and chi-squared tests were performed on grades and experience measures to find any statistically significant differences among the groups. This procedure was repeated until groups with no significant differences were found and the averages of the traditional group were better or the same as averages of the other groups for the most important properties: high school mathematics, spreadsheet usage, and programming experience. The largest difference was in programming experience ($\chi^2 = 4.054, df = 2, p = 0.3988$).

After the first lecture, 11 new students enrolled. Due to strict time limits they could not be allocated using this procedure but the groups still retained their suitability for the experiment as shown in Table 2 that summarizes the main properties of the groups. The last column of the table gives p-values from χ^2 tests and they indicate that there were no statistically significant differences between the three groups.

In spreadsheet usage, value 1 corresponds to an introductory course that all new students are supposed to take at the beginning of their studies. In programming courses, value 1 corresponds to a voluntary short introduction to programming that precedes the course of the experiment. This short introduction uses the Karel language which has, e.g., no variables. In programming experience, value 1 corresponds to having written some small programs using some programming language having variables, e.g., using Karel or HTML were not considered as programming.

The examination was a requirement of the course. Students participating in the program comprehension and program creation sessions were given a small compensation in the form of a coffee voucher. For the sessions, subjects were randomly selected among those having no or little previous background in programming.

3.1.2 Materials

The *examination* consisted of four types of questions (the number of questions in parentheses):

- Questions not related to variables (*E-NONVAR*, 2): These were used to find out possible differences among the teachers and to provide a reference point for each subject reflecting his or hers personal capabilities and amount of engagement in the course. In analyzing results, these questions were used as a "pre-test" to evaluate the scores of other questions. For this purpose, these questions were designed to test similar type of learning, i.e., a skill to apply learned materials in new situations, as the experimental questions.

  The first question concerned various looping constructs and situations for which each of them is appropriate. The second question presented syntactic rules for a strange language together with potential strings of the language. Subjects were asked which strings were legal and why.

- Program simulation (*E-SIMU*, 1): Subjects were asked to predict the output of a 15 lines long program with a given input data. The program found out prime numbers using the sieve of Eratothenes and its output contained the primes together with their accumulating sum. The program contained two *steppers*, one *fixed value*, one *gatherer*, and one *one-way flag* array; the names of the variables being meaningless one-letter identifiers.

  The use of roles was clear but the logic of the program was intended to be cumbersome (promoted by the meaningless variable names) so that students would be forced to use simulation when deciphering the output of the program.

- Program comprehension (*E-COMPR*, 1): Subjects were presented with a 19 lines long program that printed a dosage table for a week's medication, together with the total amount of medicine needed. The students' task was to "describe what is the purpose of the given program and how it works". The program had one *fixed value*, one *stepper*, one *most-recent holder*, and one *gatherer*. The variables were meaningful single letters, except the only input variable (the weight of the patient) that was a full meaningful word.

  The program had a simple logic and easily understandable domain. We expected that practically all students would understand the program and we were interested in analyzing the ways they would explain the program. Variables were named meaningfully to promote domain recognition, and to make comprehension easier. Full word identifiers were avoided to make it possible to discriminate between variable names and domain concepts in analyzing program descriptions.

- Program construction (*E-CONSTR*, 1): Subjects were asked to write a program that first gets as its input the number of exercise sessions and the total number of exercise assignments. Then, the number of accomplished assignments in each exercise session for a student will be input and the program calculates whether the student has accomplished a required number of assignments. This will be repeated as many times as there are students.

  This programming task was designed to make sense for the students attending the examination, and to call for the use of several roles. An optimal solution would use two *most-recent holders* that change to *fixed values* after the initial phase of the program, two *steppers*, one *most-recent holder*, and one *gatherer*.

Moreover, subjects were asked about how actively they had attended to lectures and exercises. Students that had attended less than 40 % of lectures or exercises were discarded from the results because the effect of the instruction to their performance was questionable.

For the *program comprehension protocol tasks (P-COMPR)* two Pascal programs with sample input and output were prepared. The first program (48 lines excluding blank and comment lines) contained no loops and it was used in the middle of the course; the second one (29 lines) was used at the end. Similarly, two programming problems with example input and output were made for the *program creation protocol tasks (P-CONSTR)*. All materials for the protocol tasks were pretested using second-year students, and small adjustments were made to improve the readability of the ready-made programs and to simplify the second programming task.

Table 3. Original grades in the experiment.

Question	Group					
	Traditional $n = 10$		Roles $n = 17$		Animation $n = 17$	
	Mean	SD	Mean	SD	Mean	SD
E-NONVAR/1	4.9	0.98	3.9	1.69	4.6	1.48
E-NONVAR/2	4.3	1.11	3.5	1.61	3.9	1.56
E-SIMU	3.4	2.21	2.5	2.20	2.8	2.36
E-COMPR	4.5	0.59	4.1	1.69	3.6	1.52
E-CONSTR	3.8	1.48	3.8	1.72	3.9	1.51

3.1.3 Procedure

The *examination* lasted four hours. Students' answers were first graded normally and then analyzed for the purposes of this experiment. The examination was graded for the purposes of the course with a maximum of 6 points for each question. All grades were checked by another teacher. Although several teachers were used for grading, all answers to each question were graded by the same first and second graders.

The *program comprehension protocol tasks* (P-COMPR) were run individually, each session lasting between 9 and 47 minutes. Subjects' task was to familiarize themseleves with the program, to summarize it verbally, and to explain the meaning of each variable.

The *program creation protocol tasks* (P-CONSTR) were run in pairs working on the same program. The purpose of this procedure was to encourage subjects to verbalize their thinking when creating the program. When a pair had finished its task, the experimenter asked them to explain the meaning of each variable. Program creation sessions lasted between 18 and 65 minutes.

3.2 Results

This paper presents the first analysis of the results obtained from the examination. The program comprehension protocol tasks (P-COMPR) and program creation protocol tasks (P-CONSTR) will be analyzed later.

Sixty subjects attended the examination. Subjects that attended less than 40 % of lectures or exercises were discarded, leaving 44 subjects for the analysis. Table 3 lists average grades and standard deviations for each question and each group. Differences between groups are non-significant for each question.

The grades of the two E-NONVAR questions behave similarly: the traditional group is best, the animation group next best, and the roles group worst in both of them. Pearson's correlation coefficient between the two grades is $r = 0.412$, the two-tailed probability for a correlation of such magnitude to occur by chance being statistically significant ($SE(b) = 0.281, t = 2.931, df = 42, p = 0.0054$).

These two questions were not related to variables in any way; so differences in grades do not depend on the independent variable – the content of instruction – but reflect variables that could not be controlled: differences between teachers, and subjects' level of engagement in the course. To compensate for these differences, we will not use the grades of Table 3 as such but we will use the difference between a subject's grade (E-SIMU, E-COMPR, E-CONSTR) and his or hers average for the two E-NONVAR grades as scores for further analysis. In order to make figures easier to read, we will furthermore scale the differences so that the average of the scores of the traditional group will be 3.0.

The scores for the *program simulation question* (E-SIMU) are presented in Figure 3. Differences between the groups are non-significant.

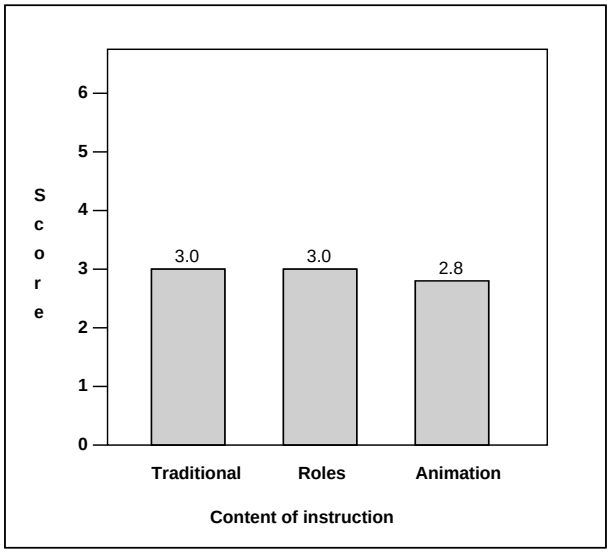

Figure 3. Average scores of the program simulation question (E-SIMU).

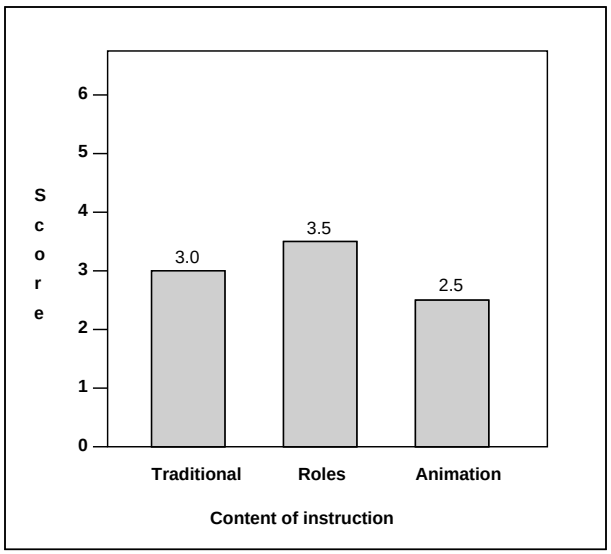

Figure 4. Average scores of the program comprehension question (E-COMPR).

The scores for the *program comprehension question* (E-COMPR) are presented in Figure 4. The difference between the roles and animation groups is significant (two-tailed t test, $t = 2.026, df = 41, p = 0.0493$).

Answers were further analyzed according to the correctness of comprehension. We selected all answers having no errors and demonstrating full understanding of every detail of the program, and counted group grade averages for them. For the traditional group the grade average was 4.6, for the roles group 5.0, and for the animation group 4.1 (roles vs. animation, two-tailed t test, $t = 1.718, df = 27, p = 0.0972$). In the examination, answers were graded not only based on the completeness of comprehension but on the "quality" – as perceived by the grading teachers – of the explanation, also. As all answers selected into this analysis demonstrated complete understanding, the differences in grade averages imply differences in the way subjects described the program.

To find out qualitative differences in subjects' descriptions, we have planned to analyze for each program description the proportions of statements according to the type of information referred to:

- *domain:* statements concerning the input-output relation and other aspects of the program related to its task from a user's perspective
- *data:* statements concerning data flow and the meaning of variables not directly visible in the program (i.e., deep structure)

Table 4. Distribution of program descriptions (E-COMPR) according to the level of expression.

Group	Level	
	Operation	Data only
Traditional	8	2
Roles	15	2
Animation	11	6

- *mixed data:* statements relating data flow and the meaning of variables not directly visible in the program with domain information
- *operation:* statements concerning specific operations and control structures directly visible in the program text (i.e., surface structure)
- *mixed operation:* statements relating operations and control structures directly visible in the program text with domain information

This classification is a simplified version of that used by Pennington (1987). Her subjects were experts working on a moderate size program whereas our subjects were novices working on a short program, and therefore we cannot expect our subjects to use as rich variety of statements as Pennington did. Pennington found that data (and mixed data) level statements reflect deep knowledge of a program and represent better comprehension than (mixed) operation level statements that are related to the surface structures of programs. When studying a program, it is impossible to form data level knowledge unless the individual operations have been understood.

For the purposes of this paper, we did, only analyze whether program descriptions contained any statements at the operation or mixed operation levels (in addition to possible domain and data level statements; column "Operation" in Table 4) or were written at domain and data level only (column "Data only"). Due to the phrasing of the question, practically all program descriptions contained domain level statements. Table 4 gives the number of subjects in each group using at least some operation level statements vs using data and domain level statements only. Data level descriptions are most common among the animation group (roles vs. animation, $\chi^2 = 2.615, df = 1, p = 0.1058$). Now the low grade average of the animation group can be explained: grading favored detailed, operation level descriptions yielding lowest grade average for the animation group that had most data level descriptions.

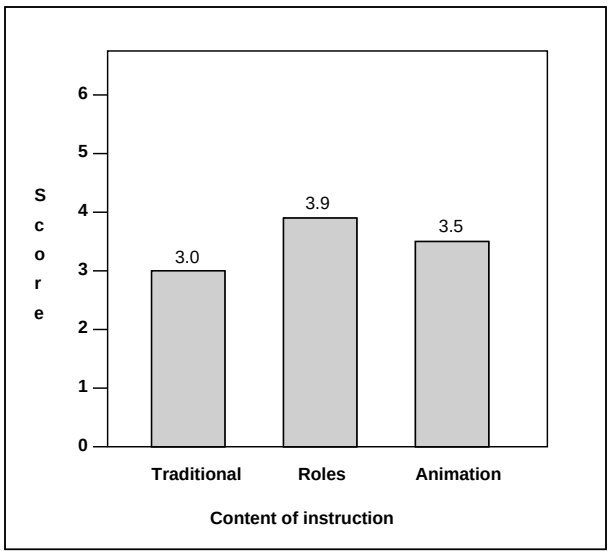

Figure 5. Average scores of the program construction question (E-CONSTR).

The scores for the *program construction question* (E-CONSTR) are presented in Figure 5. Differences between the groups are non-significant.

A qualitative analysis of errors in the programs will be done later and is not treated in this paper.

Finally, we analyzed the use of role names in the answers. As expected, no subject in the traditional group used role names. The roles and animation group behaved equally: 35 % of the subjects in both groups used role names. Roles were usually assigned correctly with the only errors made by two subjects in the roles group.

3.3 Discussion

The questions in the examination called for various programming-related activities: program simulation, program comprehension, and program creation. Even though most differences in the results are not statistically significant, the trends suggest that the effects of using roles and role-based animation depend on the nature of the activity.

In *program simulation*, the differences between the groups are smallest as compared to the other two question types. The traditional and roles groups performed equally well, while the performance of the animation group was slightly worse. The logic of the program to be simulated was complicated and cumbersome for the students, so the only way to find out its output was to simulate its execution carefully. Even though the roles of the variables were easy to find, the moments when variables were updated was not easy to predict. It is therefore natural that knowledge of roles does not help in this task.

Jehng *et al.* (1999) have studied effects of visualization on learning recursion. They found smaller differences in tasks where subjects had to predict the outcome of programs than in program creation tasks. Our results agrees with this result.

In *program comprehension*, the roles group performed best while the animation group was worst. The analysis of the program descriptions showed that the traditional and roles groups gave detailed, operation level descriptions, while data level descriptions were most common among the animation group.

This difference in the nature of the program descriptions provided by animation group subjects may be explained by the differences in the software used in exercises for program animation. A semi-structured interview with the teacher who supervised all exercise sessions of the roles and animation groups revealed that PlanAni users concentrated more on variables while debugger users spent most of their time following program code (Sajaniemi and Kuittinen, 2003). Even though PlanAni flashes each code fragment before animating its effect, students appeared not to follow the code. As a consequence, debugger users got a better understanding of the detailed actions of the code but PlanAni users got a better understanding of the total effect of the program and how each variable contributed to this. This might have affected the way PlanAni users think about programs: they may consider the life-cycles of variables more important than individual actions of the program.

An analysis of the grades in the examination revealed that the teachers gave better grades for detailed answers that explained the working of the program statement by statement than for higher-level descriptions of properties not directly visible in the program. As the descriptions of the animation group stressed data aspects more than program code, their grades were low. However, data level knowledge is an indication of superior programming skill (Détienne, 2002; Pennington, 1987). For example, Clancy and Linn (1999) cite a study demonstrating that code reuse – which demonstrates expert-like programming skill – was substantially more common for students who gave data level program summaries. As a consequence, we may deduce that the graders did give less credit for better descriptions. One may wonder, whether this behavior is common in programming teachers who usually are ignorant of even the most central results of the psychology of programming.

In *program construction*, the roles and animation groups outperformed the traditional group. Again, the scores for the roles group were better than those of the animation group. We have not yet analyzed the types of errors subjects of the three groups made, so it is impossible to suggest any explanation for this. In any case, even though the differences are not statistically significant it seems that teaching

roles to novice programmers helps them in program construction.

4. Conclusions

We have conducted an experiment to study the effects of using the roles of variables concept and role-based animation in teaching programming to novices, and presented a first analysis of the results. The role concept captures tacit expert knowledge in a form suitable to be introduced to novices. The central research question has been how the teaching of this knowledge affects novices' programming skills in program simulation, program comprehension, and program construction.

The results show that students were able to understand the role concept and to apply it in new situations: after the course, 35 % of the subjects used role names in their exam answers even though the questions did not mention roles in any way. All experimental groups performed equally well on program simulation but groups that had been introduced to roles performed better in program comprehension and construction. Moreover, the use of the role-based animator affected the way students describe programs: animator users stressed data-related issues that describe the deep structure of programs while the other groups stressed directly visible operations and control structures that represent the surface structure. Thus, the use of the animator led to descriptions that indicate better programming skills.

The deeper understanding caused by the animator was not, however, reflected in the course grades. Teachers gave better grades for detailed surface structure descriptions than for answers revealing deep understanding. It would be interesting to see whether such a behavior is common among teachers whose knowledge of the psychology of programming is usually poor.

The analysis of the answers will be continued, and the program comprehension and constructions protocols, that were not dealt with in this paper, will be analyzed, too. We hope that these activities will lead to a better understanding of how the role concept and role-based animation affect students' programming and comprehension processes.

Acknowledgments

The authors would like to thank Elina Räisänen, Markku Hauta-Kasari, Jenni Pitkänen, and Matti Niemi for acting as teachers of the course; Pauli Byckling, Pauli Harjumäki, and Veli-Pekka Laasonen for practical help in running the experiment.

References

Ben-Ari M., Sajaniemi J. (2003). Roles of Variables From the Perspective of Computer Science Educators. *Submitted.*

Clancy M. J., Linn M. C. (1999). Patterns and Pedagogy. *Proc. of the 30th SIGCSE Technical Symposium on CS Education,* ACM SIGCSE Bulletin, 31(1), 37–42.

Davies S. P. (1993). Models and Theories of Programming Strategy. *International Journal of Man-Machine Studies,* 39(2), 237–267.

Davies S. P. (1996). Display-Based Problem Solving Strategies on Computer Programming. *Empirical Studies of Programmers: Sixth Workshop,* eds. W. D. Gray and D. A. Boehm-Davis. Ablex Publishing Company, Norwood, NJ.

Détienne F. (2002). *Software Design - Cognitive Aspects.* Springer-Verlag, London.

Ehrlich K., Soloway E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. *Human Factors in Computer Systems,* eds. J. C. Thomas and M .L. Schneider. Ablex Publishing Co, Norwood, N.J., 113–133.

Feldman M. (1999). *Ada95 Problem Solving and Program Design.* Addison-Wesley.

Fincher S. (1999). What are We Doing When We Teach Programming? *Proc. of the 29th ASEE/IEEE Frontiers in Education Conference,* San Juan, Puerto Rico, Session 12a4, pages 1–5.

Fincher S., Utting I. (2002). Pedagogical Patterns: Their Place in the Genre. *Proc. of the ITiCSE'02,* ACM Press.

Fleury A. (1997). Acting Out Algorithms: How and Why It Works. *Proc. of the 4th Annual CCSC Midwestern Conference,* Dominican University.

Ginat D. (2001). Early Algorithm Efficiency with Design Patterns. *Computer Science Education,* 11(2), 89–109.

Green T. R. G., Cornah A. J. (1985). The Programmer's Torch. *Human-Computer Interaction - INTERACT'84,* IFIP, Elsevier Science Publishers (North-Holland), 397–402.

Hagan D., Sheard J. (1998). The Value of Discussion Classes for Teaching Introductory Programming. *Proc. of the ITiCSE'98,* ACM Press, 108–111.

Hanly J.R., Koffman E. B. (1999). *Problem Solving and Program Design in C.* Addison-Wesley.

Hundhausen C. D., Douglas S. A., Stasko J. D. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing,* 13, 259–290.

Jehng J-C. J., Tung S-H. S., Chang C-T. (1999). A Visualization Approach to Learning the Concept of Recursion. *Journal of Computer Assisted Learning,* 15, 279–290.

Jenkins T. (1998). A Participative Approach to Teaching Programming. *Proc. of the ITiCSE'98,* ACM Press, 125–129.

Koffman E. B. (1986). *Turbo Pascal: A Problem Solving Approach.* Addison-Wesley, Reading, MA.

McKay E. (1999*a*). An Investigation of Text-Based Instructional Materials Enhanced with Graphics. *Educational Psychology*, 19(3), 323–335.

McKay E. (1999*b*). Exploring the Effect of Graphical Metaphors on the Performance of Learning Computer Programming Concepts in Adult Learners: A Pilot Study. *Educational Psychology*, 19(4), 471–487.

Nguyen D. (1998). Design Patterns for Data Structures. *Proc. of the 29th SIGCSE Technical Symposium on CS Education*, ACM SIGCSE Bulletin, 30(1), 336–340.

Pennington N. (1987). Comprehension Strategies in Programming. *Empirical Studies of Programmers: Second Workshop*, eds. G. M. Olson, S. Sheppard, E. Soloway. Ablex Publishing Company, Norwood, NJ, 100–113.

Pirolli P. L., Anderson J. R. (1985). The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian J. Psychology*, 39(a), 240–272.

Rist R. S. (1991). Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction*, 6, 1–46.

Sajaniemi J. (2002). An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, IEEE Computer Society, 37–39.

Sajaniemi J. (2003). *Roles of Variables Home Page*. http://www.cs.joensuu.fi/~saja/var_roles/ (accessed Jan. 24th, 2003).

Sajaniemi J., Kuittinen M. (2003). Program Animation Based on the Roles of Variables. Accepted to the ACM Symposium on Software Visualization SoftVis'03, June 11-13, 2003, San Diego, USA.

Wallingford E. (2003). *The Elementary Patterns Home Page*. http://www.cs.uni.edu/~wallingf/patterns/elementary/ (accessed Jan. 24th, 2003).

Wiedenbeck S. (1989). Learning Iteration and Recursion from Examples. *Int. J. Man-Machine Studies*, 30(1), 1–22.