Using Roles of Variables in Teaching: Effects on Program Construction

Pauli Byckling and Jorma Sajaniemi

University of Joensuu, Department of Computer Science
P.O.Box 111, 80101 Joensuu, Finland,
pauli.byckling@cs.joensuu.fi,
WWW home page: http://www.cs.joensuu.fi/~pbyckli/

Abstract. Roles of variables capture 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 procedural programs. This paper presents the results from a protocol analysis of program creation tasks in 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 use of the program animator increases novices' ability to apply data-related programming plans in program construction and thus increases programming skill. Plan knowledge and use is analyzed using a new model that is based on Rist's theory of schema expansion.

1 Introduction

Programming skill is hard to acquire. 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 reorganisation of topics and introduction of new concepts are far more rare.

We know only two examples of research into *new concepts* that can be utilized in teaching elementary programming: software design patterns, and roles of variables. Software design patterns [1] 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 [2]) and application areas (e.g., data structures [3]). Research into the use of patterns indicates that instructors should expect on a regular basis to refine the patterns they offer students [1].

Roles of variables [4, 5] 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 procedural programming, and they can be described in a compact and easily understandable way [4]. Ben-Ari and Sajaniemi [6] have shown that in one hour's work, computer science teachers can learn roles and assign them successfully in normal cases. As opposed 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, a classroom experiment with three experimental conditions was conducted: 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. Sajaniemi and Kuittinen [7] analyzed the final examination of the course. They found that roles provide students a new conceptual framework that enables students to mentally process program information in a way that demonstrates good programming skills. Moreover, Sajaniemi and Kuittinen found some support for the assumption that the use of the animator fosters adoption of role knowledge. The current paper reports the results of program creation tasks conducted during the experiment.

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 followed by results in Section 4 and discussion in Section 5. Finally, Section 6 contains the conclusion.

2 Roles of Variables

Sajaniemi [4] 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 analyzing large-scale programs. His work is based on earlier studies on variable use made by Ehrlich and Soloway [8], Rist [9], and Green and Cornah [10]. Later, roles have been found to be a part of experts' programming knowledge [11] and it has been applied to object-oriented and functional programming. In this Section we will describe the role concept and its potential uses in teaching to program.

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.

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 one 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

```
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.
```

Fig. 1. A short Pascal program.

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* [5]. 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 an 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 [12] and the decision of what knowledge to apply first is not easy. This problem can be diminished by guiding novices to start a programming task by thinking about data requirements: what roles (and consequently

Table 1. Informal role definitions.

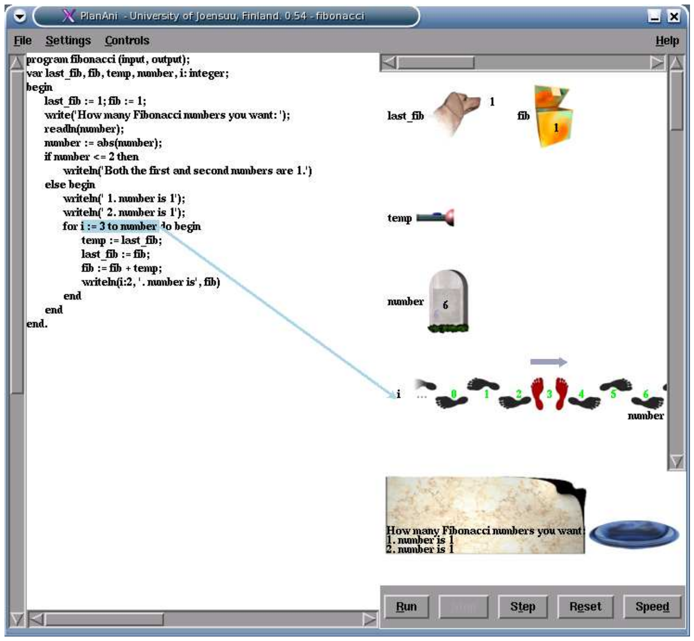
Role	Informal definition
Fixed value (FIX)	A variable which is initialized without any calculation and whose value does not change thereafter.
Stepper (STP)	A variable stepping through a succession of values that can be predicted as soon as the succession starts.
Most-recent holder (MRH)	A variable holding the latest value encountered in going through a succession of values.
Most-wanted holder (MWH)	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 (GAT)	A variable accumulating the effect of individual values in going through a succession of values.
Transformation (TRN)	A variable that always gets its new value from the same calculation from value(s) of other variable(s).
Follower (FOL)	A variable that gets its values by following another variable.
One-way flag (ONE)	A two-valued variable that cannot get its initial value once its value has been changed.
Organizer (ORG)	An array which is only used for rearranging its elements after initialization.
Temporary (TMP)	A variable holding some value for a very short time only.
Other (OTH)	Any other variable.

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 [13] 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.

The animation of operations depends on the roles, also. For example, an assignment to a follower is animated by transferring the value of the followed variable into the follower whereas the update of a stepper is animated by transferring values within the role image itself. Similarly, the animation of comparisons varies according to the role.

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 the right pane is reserved for variables, and below it there is an 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



```
program fibonacci (input, output);
var last_fib, fib, temp, number, i: integer;
begin
    last_fib := 1; fib := 1;
    write('How many Fibonacci numbers you want: ');
    readln(number);
    number := abs(number);
    if number <= 2 then
        writeln('Both the first and second numbers are 1.')
    else begin
        writeln(' 1. number is 1');
        writeln(' 2. number is 1');
        for i := 3 to number do begin
            temp := last_fib;
            last_fib := fib;
            fib := fib + temp;
            writeln(i:2, '. number is', fib)
        end
    end
end.
```

```
How many Fibonacci numbers you want:
1. number is 1
2. number is 1
```

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

the right. Whenever the color enhancement is moved to a new location in the program, the enhancement flashes.

3 Experiment

To test the hypothesis that introducing roles of variables in teaching facilitates learning programming, an experiment was conducted during an introductory Pascal programming course at university level [7]. Students attending to the course were divided into three groups that were instructed differently: one receiving normal lectures and exercises (the *traditional group*), one attending lectures with systematic use of 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*). The experiment was a between-subject design with the content of instruction as the between-subject factor. Preparations of the experiment were made with caution in order to prevent uncontrolled effects (see [7] for detailed information about the experiment).

Sajaniemi and Kuittinen [7] analyzed the *final examination* of the course. They found 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 answers even though the questions did not mention roles in any way. But the exposition to roles resulted not only in a better vocabulary; a more important effect was that roles provided students a new conceptual framework that enabled them to mentally process program information in a way similar to that of good code comprehenders. The use of role-based animation seemed to foster the adoption of role knowledge as animation users had less problems with variables in program construction. Moreover, animation users tended to stress deep program structures which is a sign of better comprehension. Thus, both the use of roles in teaching and the use of role-based animation led to results that indicate better programming skills.

In the middle and at the end of the course, some students were given *program construction tasks* which were videotaped. Speech protocols were transcribed from the videotapes of the tasks. This paper presents results of the two tasks, *P-CONSTR-MID* in the middle of the course and *P-CONSTR-END* at the end of the course.

Subjects Subjects were undergraduate students studying computer science for the first semester. They were randomly selected among those attending the course and having no or little previous background in programming. In total, 22 students attended the first program construction task: 6 students from the traditional group, 6 students from the roles group and 10 students from the animation group. In the latter task the amount of subjects was 20: again 6 students both from the traditional group and from the roles group and 8 students from the animation group. Students participating in the sessions were given a small compensation in the form of a coffee voucher.

Materials For both program construction tasks a single elementary level programming problem was made. Complexity of these problems reflected the prevailing state of the course. Thus the first task was more trivial with no loops or complex role plans needed in the solution, whereas the latter task was more challenging.

The programming assignment in P-CONSTR-MID was to calculate number of men in Scandinavian countries using a percentage value and populations of countries given as input. In P-CONSTR-END the assignment was to read times in hours and minutes repeatedly and to transform each input pair into minutes only. In addition, time closest to one hour had to be found as well as whether any of the given times was exactly two hours. Both tasks consisted of two separate parts, part A (basic functionality) and part B (supplementary functionality). All materials for the protocol tasks were pretested using second-year students, and small adjustments were made to simplify the second programming task. The final versions can be found at

http://cs.joensuu.fi/˜saja/var_roles/literature.html.

Procedure The program creation tasks were run on computer terminal using Turbo Pascal programming environment. We used pairs of students working together. The purpose of this procedure was to encourage subjects to verbalize their thinking when

Table 2. Number of variables used by each pair in P-CONSTR-MID.

Pair	FIX	TRN	GAT	Total
TRAD-M1	6	0	0	6
TRAD-M2	6	0	1	7
TRAD-M3	6	5	0	11
ROLE-M1	6	4	0	10
ROLE-M2	6	1	1	8
ROLE-M3	6	5	0	11
ANIM-M1	6	1	2	8
ANIM-M2	6	0	2	7
ANIM-M3	6	0	0	6
ANIM-M4	6	0	1	7
ANIM-M5	6	0	0	6
OPTIMAL	1	1	1	3

creating the program. Both parts of the assignments were given in the same session with no pause between them. Depending on the implementation of part A, implementation of part B caused modifications to existing program code in some cases. Program creation sessions lasted between 18 and 65 minutes. If subjects tried to give up with an incomplete solution before the time limit of one hour was reached, they were encouraged to give a better try.

4 Results

Sections 4.1 and 4.2 present the properties of the final programs: the use of variables and correctness of solutions. The main analysis—expression of plan development in the implementation of variables—is presented in Section 4.3. The data for this analysis consisted of program writing protocols collected from the videos and of the transcribed speech protocols. Analyses of the results were made by the first author of this paper.

4.1 Use of variables

In P-CONSTR-MID, variables in the optimal implementation are a fixed value, a most-recent holder, and a gatherer. In P-CONSTR-END, the optimal set of variables comprises two most-recent holders, one transformation, one most-wanted holder, and a one-way flag. Use of variables by each pair of students is presented in Tables 2 and 3. Each pair is referred to by the name of the group followed by an abbreviation of the task (M for MID; E for END) and pair number, e.g., TRAD-M1 refers to the first pair of the traditional group in P-CONSTR-MID.

In P-CONSTR-MID differences between groups were small and no patterns could be identified. Thus, the rest of this paper concentrates on the latter task.

Due to the small sample size in P-CONSTR-END (10 pairs), no statistical significance tests could be made. However, frequencies in Table 3 show a distinctive difference between the animation group and the two other groups. While in the traditional and in

Table 3. Number of variables used by each pair in P-CONSTR-END.

Pair	MRH	TRN	MWH	ONE	STP	OTH	Total	Full coverage
TRAD-E1	2	1	-	-	-	-	3	no
TRAD-E2	2	2	2	2	-	-	8	yes
TRAD-E3	2	1	-	-	1	-	4	no
ROLE-E1	2	1	-	-	-	-	3	no
ROLE-E2	2	1	2	1	-	1	7	yes
ROLE-E3	2	2	1	-	-	-	5	no
ANIM-E1	2	1	1	1	-	-	5	yes
ANIM-E2	2	1	1	1	-	-	5	yes
ANIM-E3	2	1	1	1	-	-	5	yes
ANIM-E4	2	1	1	1	-	-	5	yes
OPTIMAL	2	1	1	1	-	-	5	

the roles group there is a variety of variables used (traditional: 3–8, roles: 3–7), each pair in the animation group used the optimal set of variables. The column ’Full coverage’ presents whether the pair has implemented all five variables needed in the task. However, this is not an indication of fully correct implementation. The single variable considered as ’OTH’ is a variable which was declared but not used.

4.2 Correctness

Programs were next analyzed according to their semantical correctness. For this analysis we divided the latter task (P-CONSTR-END) into three subtasks and then examined the correctness of each subtask separately.

Subtask *T1* is directly part A of the assignment consisting of the reading of input, implementation of a loop, calculation and output functions. Subtask *T2* contains the comparison of inputs in order to find time closest to one hour and storage of the best value. Third subtask, *T3*, comprises checking if the entered time is exactly two hours and storing the information whether the condition is met. The results are presented in Table 4. Definitions for the error abbreviations are introduced in Table 5. The animation group performed best, having 11 correct subtasks out of 12 (92 %). The roles group and the traditional group performed evenly, both having 4 correct subtasks out of 9 (44 %).

All actual errors are in subtasks T2 and T3. Percentage of correctness in overall is 30 % in T2 and 60 % in T3. In these subtasks the animation group has 7/8 correct (88 %) whereas the same ratio in the two other groups combined is 2/12 (17 %). In T2, none of the pairs from the traditional nor from the roles group reached correct outcome.

4.3 Role plan development

The preceding results are based on an analysis of the final programs. In this Section we analyze role plan development by examining the writing order of program code lines observed in the videos. The basis of this analysis is Rist’s model of schema creation in

Table 4. Correctness of subtasks in P-CONSTR-END.

Pair	T1	T2	T3
TRAD-E1	ok	–	–
TRAD-E2	ok	iw	ok
TRAD-E3	ok	–	ar
ROLE-E1	ok	–	–
ROLE-E2	ok-	i-	ok
ROLE-E3	ok	uw	–
ANIM-E1	ok	ok	ok
ANIM-E2	ok	ok	ok
ANIM-E3	ok	ok	ok
ANIM-E4	ok	i-	ok

Table 5. Notation used in Table 4.

Symbol	Interpretation
ok	correct implementation
ok-	acceptable but obscure implementation
ar	variable replaced with an array
–	missing totally
i-	missing initialization
iw	variable intialized with wrong value
uw	improper variable update(s)

programming [9, 14]. According to Rist, programming process consists of implementation of *program plans*, which either have to be retrieved from memory, or created. A program is designed by the process of *schema expansion*, showing a pattern of forward design, if the process is guided by existing knowledge and program plans are retrieved from memory. Furthermore, this forward design continues all the way to the level of program code so that the corresponding program code is written top down, i.e., in the order it appears in the final program. On the other hand, if plans are created during programming, the development will show a pattern of local backward design, called as *focal expansion*.

The analysis model According to Rist [14], the development of a program can be traced by defining the plans used in the program and noting the order in which pieces of the plans appear during programming. Rist introduces a few basic plans (prompt plan, label plan, running total plan, found plan, and loop plan) which he divides into three specific pieces: Initialization (I), Calculation (C) and Output (O).

Roles of variables can be considered as a certain kind of programming plans but Rist's model does not apply to roles as such. For roles a larger number of plan pieces are required: *Goal (G)* (can be found from the speech protocol only), *Declaration (D)*, *Initialization (I)*, *Extension (E)*, *Computation (C)*, *Use of the latest value (U1)* and *Use of the final value (U2)*. Table 6 lists actions which execute these pieces for each role.

The model is not restricted to the programs in the current experiment but it is generally applicable when examining role plans in program creation.

The major difference between Rist's model and our model is in the scope of examination. Rist uses the whole program code in the analysis, but our main interest is in the use of variables, so each variable is analyzed individually. The data consists only of lines which are directly related to the role plan accompanied with complementary fractions from the speech protocol in some cases.

Scoring When deciding whether the emergence of a plan represents schema expansion or focal expansion, Rist looked at the order in which plan pieces are written during programming. In focal expansion (backward development) code development proceeds from the calculation—so called focal line—to initialization. In schema expansion (forward development) the chronological appearance order of plan pieces reflect the final form of the plan schema in program code.

In Rist's model, the decision is based upon the plan components I and C only, C-I meaning focal and I-C schema expansion. In small programs, schema expansion can, however, be expected to cover other plan pieces, also. On the other hand, small programs may also be written obeying the final textual order even though this order of plan pieces may sometimes differ from the theoretical order G-D-I-E-C-U1-U2. Therefore, we will consider plan emergence to represent forward development (FD) if the appearance order of plan pieces follows either the theoretical order of the role plan or the order of the final program code. Other orders are considered to reflect backward development (BD). With incomplete plans, missing pieces are scored as though they would appear later.

In our scoring, forward development represents schema expansion. In backward development, either the calculation or the use of a variable emerge earlier than expected; this represents either focal or use expansion respectively.

Example data in Table 7 will illustrate the analysis. The chronological appearance order of these plan pieces (i.e., lines of code) is D-C-E-U1. The order of these lines in the final program is D-E-C-U1, which is also the theoretical order, thus the example shows a pattern of backward development.

As Table 4 shows, some pairs are lacking subtasks T2 or T3. In scoring, variables needed in implementing these functions were scored as ”no development” (ND). In the analysis NDs are regarded as backward development. As the task consisted of two separate parts, in some cases implementation of the latter part caused some modifications to the existing code. In the analysis only the first implementation of a variable has been taken into account.

The frequencies of FD scores for the latter program creation task (P-CONSTR-END) are presented in Figure 3.

5 Discussion

In overall, there were only three fully functional implementations out of ten programs in P-CONSTR-END, all in the animation group. The animation group performed best in all analyses (use of variables, correctness, role plan development) while the traditional group was the worst. Most substantial differences were in the most complex subtasks.

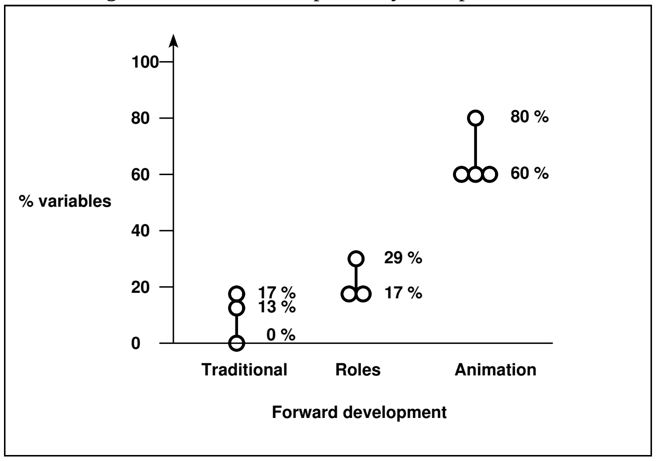
Table 6. Actions in the role analysis model.

Role	Declaration	Initialization	Extension	Computation	Use 1 (latest value)	Use 2 (final value)
FIX	declaration	possible prompt to user	-	assingment of the value or reading of the input	-	use
STP	declaration	possible initialization with initial value	-	update in the loop	use in the loop	use after the loop
MRH	declaration	possible assignment of the first value or reading the value before loop (depending on the design of the loop)	implementation of the loop (possible use in controlling the loop)	reading of the value in loop	use in the loop	use of the final value after the loop
MWH	declaration	initialization (typically with first value read)	comparison between the old value and a new value	update if better value is found	use of the latest best value	use of the final best value
GTH	declaration	initialization (typically with 0)	-	update	use in the loop	use of the final accumulated value after the loop
ONE	declaration	initialization (typically with ’false’ or 0)	check of the condition which affects the flag	update if the condition is met	use of the flag (e.g. in controlling the loop)	use of the final state of the flag
TRN	declaration	-	-	update	use	-
FOL	declaration	initialization with first value (e.g. first value in a list)	-	update with the next value	use	-
TMP	declaration	-	-	assignment of the value	”giving away” the value	-
ORG	declaration (possible initialization)	filling of the array in a loop (if not filled in declaration)	-	swap of values of two variables in a loop	-	use

Table 7. Chronological development of the variable 'hours' in a programming protocol. Variable names are translations from Finnish.

Line number	Program code	Piece type
3.	var hours, minutes: integer;	Declaration
9.	readln(hours);	Computation
8.	while (hours $>$ 0) or (minutes $>$ 0) do	Extension
11.	transformation := 60 * hours + minutes;	Use 1 (latest value)

Fig. 3. Percentages of forward development by each pair in P-CONSTR-END.



All students had been given the appropriate programming knowledge in order to solve the task and both the animation group and the roles group had the same theoretical role knowledge. Still, students from the animation group were much more able to apply their knowledge adequately in program construction.

In P-CONSTR-END the animation group showed mostly forward plan development (overall FD: 65 %) whereas in the roles group there is considerably less forward development (overall FD: 21 %). In the traditional group the amount of forward development is only 10 %. Moreover, the forward development in these latter two groups concentrated on simple plans. In the two most complex plans needed in the task, the amount of forward development is 0 % both in the traditional group and in the roles group. Overall, the animation group is the only one showing forward development through all variables. This means that students in the animation group possessed and were able to apply programming schemas better than the other students.

Subjects in the animation group still had some typical novice level problems in their programming processes like others, but their problems concerned mainly programming language knowledge, i.e., details of implementation. They had considerably less problems in progamming knowledge than the students in the other two groups.

The poorer performance of the roles group as compared to the animation group can be partly explained by a confusion about roles in the roles group. The speech protocols revealed that students in the animation group explicitly used role knowledge when designing their program code (ANIM-E1: ”Let’s make it a one-way flag”; ANIM-E2: ”Well - if some value happens to be two hours then it’ll be like a one-way flag over here.”) With the roles group the speech protocols reveal that the role knowledge remained too abstract for the students to apply it successfully in problem solving (ROLE-E1, Subject B: ”Mmm. Then we should do then do a stepper which then...”; Subject A: ”Or then a holder for each time.”; Subject B: ”Yes. True. Do you remember this? It is the temporary then, isn’t it?”; ROLE-E2, Subject A: ”There has to be some... some most-wanted holder which holds the... like the best value”; Subject B: ”Well, yes”; Subject A: ”and if it’s closer to sixty, it then keeps it”; Subject B: ”Yes, I suppose so, but I just can’t get it into my head how it’s done (laughing); Subject B: ”I roughly kind of know how it should be done, but I dont know how it can be placed into this code.”)

These results reflect the same tendencies as the results presented by Sajaniemi and Kuittinen [7]. They found that groups that had been introduced to roles outperformed the traditional group in program construction. Moreover, the animation group had least problems with variables and made less errors that could be explained by poor plan knowledge. All these results were, however, statistically insignificant and the analysis was based on weak evidence that considered possible reasons for errors in final programs. Our current results are more solid and based on the amount of forward development in the programming protocols. The current analysis reveals a distinct difference between the roles group and the animation group. The optimal selection of variables and the frequent appearance of forward development in the animation group is a clear indication of superior plan knowledge and ability to apply it.

In the program comprehension task, Sajaniemi and Kuittinen found a statistically significant difference in program summary types between the traditional group on one hand and the roles and animation groups on the other hand, and they concluded that the roles and animation groups possessed a better conceptual framework for processing program knowledge. The roles group gave more detailed, low-level summaries than the animation group but the difference was small and statistically insignificant. Thus the major difference in the program comprehension task was between the traditional group on one hand and the roles and animation groups on the other hand. The current analysis indicates a major difference between the roles group and the animation group in program construction which is a more demanding task than program comprehension. This is in line with the findings of Mayer [15, p. 72–76] who found that the difference between learning outcomes from text vs. text and animation is larger in problem-solving tasks than in recall of the learned material. In our case, the (textual) presentation of roles is sufficient for program comprehension but animation is needed to elaborate role knowledge so that it can be fluidly applied in the harder task of program construction.

6 Conclusion

We have analyzed the program creation protocol tasks of an experiment studying the effects of the roles of variables and role-based animation in teaching programming to novices. The results show that the role concept had considerable effects on programming knowledge. Students who were familiar with the concept possessed more schemas and performed better. Subjects in the animation group applied forward development to all variables whereas the two other groups showed only backward development in more complex subproblems.

The earlier analysis done by Sajaniemi and Kuittinen [7] showed that in program comprehension the mental models of both the roles group and the animation group were better than those of the traditional group. Thus the increased programming knowledge—given in the form of roles—enhanced the construction of program knowledge in program comprehension. The current analysis showed that in program creation only the animation group performed well. This suggests that the animator elaborated the increased programming knowledge so that the students could use it successfully also in program construction. This, along with the best performance of the animation group in all the analyses suggests that role knowledge should be elaborated by role-based program animation in introductory-level teaching.

The analysis of plan development was done using a new analysis model. The model is based on Rist's theory of programming plans and schema expansion [14] but it is extended to cover a larger number of plan components. Moreover, it includes explicitly a goal component that cannot be seen in the program code but is observable in the speech protocol only. The new model is based on the specific properties of roles, and is general in the sense that it can be applied in analyzing role schema usage in the construction of arbitrary programs.

Acknowledgments

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

This work was supported by the Academy of Finland under grant number 206574.

References

1. Clancy, M.J., Linn, M.C.: Patterns and pedagogy. In: Proc. of the 30th SIGCSE Technical Symposium on CS Education, Vol. 31 of ACM SIGCSE Bulletin (1999) 37–42
2. Wallingford, E.: The elementary patterns home page. http://www.cs.uni.edu/˜wallingf/patterns/elementary/ (2003) (Accessed Jan. 24th, 2003).
3. Nguyen, D.: Design patterns for data structures. In: Proc. of the 29th SIGCSE Technical Symposium on CS Education, Vol. 30 of ACM SIGCSE Bulletin (1998) 336–340
4. Sajaniemi, J.: An empirical analysis of roles of variables in novice-level procedural programs. In: Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02), IEEE Computer Society (2002) 37–39
5. Sajaniemi, J.: Roles of variables home page. http://www.cs.joensuu.fi/˜saja/var_roles/ (2004) (Accessed Dec. 22th, 2004).
6. Ben-Ari, M., Sajaniemi, J.: Roles of variables from the perspective of computer science educators. In: The 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004), Association for Computing Machinery (2004) 52–56

7. Sajaniemi, J., Kuittinen, M.: An experiment on using roles of variables in teaching introductory programming. Computer Science Education **15** (2005) 59–82
8. Ehrlich, K., Soloway, E.: An empirical investigation of the tacit plan knowledge in programming. In Thomas, J.C., Schneider, M.L., eds.: Human Factors in Computer Systems. Norwood, NJ: Ablex Publishing Company (1984) 113–133
9. Rist, R.S.: Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. Human-Computer Interaction **6** (1991) 1–46
10. Green, T.R.G., Cornah, A.J.: The programmer's torch. In: Human-Computer Interaction - INTERACT'84, IFIP, Elsevier Science Publishers (North-Holland) (1985) 397–402
11. Sajaniemi, J., Navarro Prieto, R.: Roles of variables in experts' programming knowledge, Accepted to the the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005) (2005)
12. Davies, S.P.: Models and theories of programming strategy. International Journal of Man-Machine Studies **39** (1993) 237–267
13. Sajaniemi, J., Kuittinen, M.: Program animation based on the roles of variables. In: ACM 2003 Symposium on Software Visualization (SoftVis 2003), Association for Computing Machinery (2003) 7–16
14. Rist, R.S.: Schema creation in programming. Cognitive Science **13** (1989) 389–414
15. Mayer, R.E.: Multimedia learning. Cambridge University Press, U.K. (2001)