An Experiment on Using Roles of Variables in Teaching Introductory Programming

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

Corresponding author:
Jorma Sajaniemi
E-mail: Jorma.Sajaniemi@Joensuu.Fi
Tel: + 358 13 2517933, Fax: + 358 13 2517955

Running title: Roles of Variables

Journal: Computer Science Education, Year: 2005
Volume: 15, Number: 1, Pages: 59–82
March 15, 2005

Abstract

Roles of variables is a new concept that captures tacit expert knowledge in a form that can 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 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 show that students are not only able to understand the role concept and to apply it in new situations but—more importantly—that roles provide students a new conceptual framework that enables them to mentally process program information in a way demonstrating good programming skills. Moreover, the use of the animator seems to foster the adoption of role knowledge.
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 effective 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 and Sheard, 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, 1999a,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 introductory 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 (2004) have shown that in one hour’s work, computer science educators 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, we conducted a classroom 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 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. The next section describes the role concept and its potential uses in teaching to program. Then, the experiment is presented and its results discussed. Finally, the last section contains the conclusions.
Roles of Variables

Sajaniemi (2002) has introduced the concept of the roles of variables which he obtained 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 (1989) 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 (2004).

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 the value of a variable is further 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.
Table 1: Informal role definitions.

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

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 succession 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 of all the programs in three introductory 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.

### Using Roles in Teaching

The set of roles is so small that it can be fully covered in an introductory programming course. Because 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 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. Using the role set, a teacher may, however, guide 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 pointing out the current action. The upper part of the right pane is reserved for 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.

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
Figure 2: The user interface of the PlanAni program animator. Variables are depicted by role images, e.g., a *stepper* (1) is represented by footprints comprising previous, current, and future values; a *fixed value* array (*candidate*) is represented by stones making it clear that the values of its elements will not change.
in giving lectures to undergraduate students and both had taught the course before. The teacher giving traditional lectures was not acquainted with 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 detect any differences caused by different teachers and students’ different degrees of engagement in the course, the examination included control 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 construction 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.

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 normal lectures and exercises (traditional group), one attending lectures with systematic use of variable roles throughout the course (roles group), and one attending the same lectures as the roles group but using role-based animator in exercises (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, declarations of variables were accompanied with their roles in comments. Students were also given a printed list describing all roles with short program examples (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. The lectures were based on existing materials not specially designed for the introduction of roles; this decision was based on our intention not to interfere with the teaching of the traditional group and to present to all groups as similar teaching as possible.

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 tried to make sure that the cognitive activities were equivalent in each group so that differences in performance would not be due to differences in cognitive activities but due to differences in the content of the visualizations.

**Subjects**

The subjects were undergraduate students studying computer science for the first semester. All 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 means of the traditional group were better or the same as means of the other groups for properties we considered most important to predict learning programming. Wilson and Shrock (2001) have studied several factors and found that comfort level, math background and formal programming training had positive influence on success in an introductory programming course. Comfort level cannot be measured at the beginning of a course, and in our case previous programming courses were practically the same for all students. Hence we selected high school math-
ematics, spreadsheet creation, and programming experience as the most important properties. The largest difference between the groups 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. However, 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 $\chi^2$ tests and they indicate that there were no statistically significant differences between the three groups.

In spreadsheet creation, 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.

Materials

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

- Control questions not related to variables ($E\text{-NONVAR}$, 2): These were used to find out possible differences among the teachers and to provide a reference point for each subject reflecting his or her 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 as the experimental questions, i.e., a skill to apply learned materials in new situations.

  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 ($E\text{-SIMU}$, 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 Eratosthenes and its output contained the primes together with their accumulating sum.
Table 2: Basic data about the experimental groups. In all scales higher values are better.

<table>
<thead>
<tr>
<th>Property</th>
<th>Traditional</th>
<th>Roles</th>
<th>Animation</th>
<th>All</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of subjects</td>
<td>26</td>
<td>32</td>
<td>33</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Female subjects (%)</td>
<td>30.8</td>
<td>18.9</td>
<td>24.2</td>
<td>24.2</td>
<td>0.7380</td>
</tr>
<tr>
<td>High school mathematics (mean, scale 1-3)</td>
<td>2.4</td>
<td>2.2</td>
<td>2.1</td>
<td>2.2</td>
<td>0.1343</td>
</tr>
<tr>
<td>High school mother tongue (mean, scale 1-3)</td>
<td>2.4</td>
<td>2.2</td>
<td>2.1</td>
<td>2.2</td>
<td>0.1851</td>
</tr>
<tr>
<td>High school information technology (mean, scale 1-3)</td>
<td>2.5</td>
<td>2.6</td>
<td>2.6</td>
<td>2.6</td>
<td>0.9828</td>
</tr>
<tr>
<td>High school art (mean, scale 1-3)</td>
<td>2.2</td>
<td>2.3</td>
<td>2.4</td>
<td>2.3</td>
<td>0.7039</td>
</tr>
<tr>
<td>Spreadsheet creation (mean, scale 0-2)</td>
<td>1.3</td>
<td>1.0</td>
<td>1.2</td>
<td>1.1</td>
<td>0.2827</td>
</tr>
<tr>
<td>Programming courses (mean, scale 0-2)</td>
<td>0.8</td>
<td>0.9</td>
<td>1.0</td>
<td>0.9</td>
<td>0.9539</td>
</tr>
<tr>
<td>Programming experience (mean, scale 0-2)</td>
<td>0.8</td>
<td>0.5</td>
<td>0.8</td>
<td>0.7</td>
<td>0.4566</td>
</tr>
</tbody>
</table>
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", i.e., to write a program summary. 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 an 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 summaries.

- 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 given as input and the program has to calculate 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 to report how actively they had attended to lectures and exercises. Students that had attended less than 40 % of
Table 3: Original grades in the experiment.

<table>
<thead>
<tr>
<th>Question</th>
<th>Traditional</th>
<th>Roles</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>SD</td>
<td>Mean</td>
</tr>
<tr>
<td>E-NONVAR/1</td>
<td>4.9</td>
<td>0.98</td>
<td>3.9</td>
</tr>
<tr>
<td>E-NONVAR/2</td>
<td>4.3</td>
<td>1.11</td>
<td>3.5</td>
</tr>
<tr>
<td>E-SIMU</td>
<td>3.4</td>
<td>2.21</td>
<td>2.5</td>
</tr>
<tr>
<td>E-COMPR</td>
<td>4.5</td>
<td>0.59</td>
<td>4.1</td>
</tr>
<tr>
<td>E-CONSTR</td>
<td>3.8</td>
<td>1.48</td>
<td>3.8</td>
</tr>
</tbody>
</table>

lectures or exercises were discarded from the results because the effect of the instruction to their performance was questionable.

The experimental materials can be found in http://www.cs.joensuu.fi/~saja/var_roles/literature.html.

Procedure

The examination, which was a requirement of the course, lasted four hours. Students’ answers were first graded normally and then analyzed for the purposes of this experiment. The maximum grade was 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. Further analysis of the results was made by the authors. Neither the graders nor the researchers making the analysis did know which group the students came from.

Results

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 mean grades and standard deviations for each question and each group. Differences between groups are non-significant for all questions.

Control questions

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 her mean 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 mean of the scores of the traditional group will be the original grade mean.

These differences between the groups must also be taken into account when considering results that cannot be compensated in this simple way. For example, percentual proportions of various types of statements within program summaries cannot be compensated, and care must be taken when interpreting the results.

**Program simulation**

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

**Program comprehension**

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, \( t = 2.026, df = 41, p = 0.0493 \)).

To study this difference, answers were analyzed with respect to the correctness of comprehension. We selected all answers having no errors and demonstrating full understanding of every detail of the program, and calculated group grade means for them. For the traditional group the grade mean 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 summary, also. As all answers selected into the new analysis demonstrated complete understanding of the program, the differences in grade means imply differences in the way subjects described the program.
Figure 3: Mean scores (and standard deviations) of the program simulation question (E-SIMU). Scores are obtained by compensating original grades by control questions grades.

![Bar chart showing mean scores of the program simulation question (E-SIMU)](chart1.png)

Figure 4: Mean scores (and standard deviations) of the program comprehension question (E-COMPR).

![Bar chart showing mean scores of the program comprehension question (E-COMPR)](chart2.png)
Table 4: Mean percentages of information type statements in program comprehension answers (E-COMPR).

<table>
<thead>
<tr>
<th>Information type</th>
<th>Traditional</th>
<th>Roles</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>10.1</td>
<td>8.2</td>
<td>9.6</td>
</tr>
<tr>
<td>Actions</td>
<td>20.4</td>
<td>18.1</td>
<td>21.2</td>
</tr>
<tr>
<td>Operations</td>
<td>11.9</td>
<td>13.2</td>
<td>11.9</td>
</tr>
<tr>
<td>State-high</td>
<td>0.4</td>
<td>2.1</td>
<td>0.0</td>
</tr>
<tr>
<td>State-low</td>
<td>0.2</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>Data</td>
<td>26.9</td>
<td>25.3</td>
<td>30.9</td>
</tr>
<tr>
<td>Control</td>
<td>6.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Elaborate</td>
<td>22.3</td>
<td>26.1</td>
<td>18.6</td>
</tr>
<tr>
<td>Meta</td>
<td>2.0</td>
<td>0.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Previous literature (e.g., Pennington, 1987; Good, 1999) indicates that qualitative differences in program summaries are related to the quality of understanding. Hence, we analyzed all program summaries (including those with errors) further. For this analysis, we used Good’s program summary analysis scheme (Good, 1999) consisting of two classifications: one based on information types and the other based on object descriptions. The information types classification is a more finely-grained and fully specified refinement of Pennington’s scheme (Pennington, 1987), while the object classification is a more restricted version of Pennington’s level of detail. Pennington used her scheme with experts working on a moderate size program, whereas our subjects were novices working on a short program—a group similar to that of Good’s subjects.

The information types classification is used to code summary statements on the basis of the information types they contain. Table 4 contains the distribution of information type statements for each of the experimental groups. (One subject from the roles group was excluded from this analysis: he was mostly copying the program code, and his behaviour was very different from any other subject threatening statistical analysis.) For detailed definitions of the information types, see (Good, 1999).

The means were tested using ANOVA; the only significant difference was in state-high ($F(2, 40) = 3.028, p = 0.0596$) containing summary statements describing the current state of the program at a more abstract level than simple test outcomes. The roles group used more state-high statements than the other groups but the absolute numbers of statements were small. In the
roles group, 31.3% of the subjects used state-high statements while in the traditional group only 10.0% of the subjects, and in the animation group none subject did use them.

The largest absolute difference was in elaborate statements, i.e., further information about something that has already been described. The roles group used more elaborate statements than the other groups, but this difference is not statistically significant ($F(2, 40) = 1.032, p = 0.3654$).

Good’s other classification, object descriptions, looks at the way in which objects are described. Each reference to an object is classified to be one of the following:

- **program only**: reference to an item which occurs only in the program domain, e.g., “counter”
- **program**: reference described in program terms, e.g., a variable identifier
- **program – real-world**: reference using terminology which is valid in both real-world and program domains, e.g., “number”
- **program – domain**: a mixture of program and problem domain references, e.g., “list of heights”
- **domain**: reference using domain terms, e.g., “sunny days”
- **indirect reference**: an anaphoric reference, e.g., “it”
- **unclear**: an ambiguous reference that cannot be coded

Table 5 contains the distribution of object description statements for each of the experimental groups.

The roles and animation groups used more program only statements ($F(2, 40) = 1.791, p = 0.1800$; traditional vs. animation two-tailed $t$ test, $t = 1.798, df = 40, p = 0.0797$), mostly because they had a rich vocabulary for role names as opposed to the traditional group’s vocabulary consisting of “counter” and “loop counter” only. On the other hand, program statements were the least common among the animation group, but this difference is not statistically significant ($F(2, 40) = 1.012, p = 0.3725$).

Even though the group means in Table 5 are similar, there were major differences in the distributions of statements within groups: program summaries in the traditional group had either a small or a large number of domain statements while in the other two groups domain statements were used
Table 5: Mean percentages of object description statements in program comprehension answers (E-COMPR).

<table>
<thead>
<tr>
<th>Object description type</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional</td>
</tr>
<tr>
<td>Program only</td>
<td>0.8</td>
</tr>
<tr>
<td>Program</td>
<td>27.1</td>
</tr>
<tr>
<td>Program – Real-world</td>
<td>0.2</td>
</tr>
<tr>
<td>Program – Domain</td>
<td>11.3</td>
</tr>
<tr>
<td>Domain</td>
<td>49.9</td>
</tr>
<tr>
<td>Indirect reference</td>
<td>5.1</td>
</tr>
<tr>
<td>Unclear</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Figure 5: Distribution of subjects with different program summary types (E-COMPR). Summary type is determined by the amount of domain and program information in object descriptions.
more evenly. To analyze this difference further, we used a strategy similar to Pennington (1987) and sorted program summaries into three types depending on the amount of domain vs. program statements in object descriptions. Summaries with at least 67% of domain statements (indirect and unclear statements excluded) were called \textit{domain-level summaries}, summaries with at least 67% of program and program only statements were classified as \textit{program-level summaries}, and all others were called \textit{cross-referenced summaries} because they had a more even distribution of domain and program information. Figure 5 depicts the distribution of subjects in each group based on the type of the program summary they gave. The number of cross-referenced summaries was significantly smaller among the traditional group than among the other groups ($\chi^2 = 10.773, df = 2, p = 0.0046$). In Pennington’s study, high comprehension programmers almost uniformly used cross-referenced summaries while low comprehension programmers tended to produce either a program-level summary or a domain-level summary. Thus the program summaries of the roles and animation groups were similar to those of high comprehension programmers.

This analysis does not, however, explain the differences in the grades of corrects answers. To study this further we looked at the correlation between grades and program summary properties of answers demonstrating complete or almost complete understanding. In addition to information type and object description contents, summary length and correctness were also included in this analysis. (Summary length means were 152.1 words for the traditional group, 154.4 for the roles group, and 141.4 for the animation group.)

Table 6 lists all statistically significant correlations. The largest correlations are negative and concern summarizing high-level features of programs: summaries with a higher proportion of action statements and domain terminology got worse grades while summaries with lower-level operation statements and program terminology got better grades. Even the length of summaries and the amount of repetition in answers correlate more with the grade than the correctness of the summaries.

The last column in Table 6 tells whether the roles group or the animation group had a larger mean for the property in question. It is interesting to note that for almost all properties having positive correlation with the grade, the roles group had a larger mean than the animation group, and vice versa. Even though correlation does not mean causality, the differences in the grades are now easy to explain: teachers grading the summaries favored summary properties that were more common among the roles group (i.e., low-level information and perhaps wordiness) and disfavored properties that
Table 6: Pearson’s correlations of program comprehension (E-COMPR) grades and program summary properties. The last column indicates whether the mean of the roles group was smaller (‘<’), larger (’>’), or equal (”=”)
to that of the animation group

<table>
<thead>
<tr>
<th>Summary property</th>
<th>Correlation</th>
<th>p</th>
<th>Roles vs. Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information type / Actions</td>
<td>-0.763</td>
<td>0.000</td>
<td>&lt;</td>
</tr>
<tr>
<td>Object description / Domain</td>
<td>-0.731</td>
<td>0.000</td>
<td>&lt;</td>
</tr>
<tr>
<td>Summary length</td>
<td>+0.629</td>
<td>0.000</td>
<td>&gt;</td>
</tr>
<tr>
<td>Object description / Program</td>
<td>+0.548</td>
<td>0.001</td>
<td>&gt;</td>
</tr>
<tr>
<td>Information type / Elaborate</td>
<td>+0.544</td>
<td>0.001</td>
<td>&gt;</td>
</tr>
<tr>
<td>Object description / Unclear</td>
<td>+0.414</td>
<td>0.012</td>
<td>&lt;</td>
</tr>
<tr>
<td>Correctness of summary</td>
<td>+0.413</td>
<td>0.012</td>
<td>=</td>
</tr>
<tr>
<td>Information type / Operations</td>
<td>+0.395</td>
<td>0.017</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

Table 7: Percentages of subjects making errors in program construction (E-CONSTR) sorted by required knowledge type.

<table>
<thead>
<tr>
<th>Required knowledge type</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional</td>
</tr>
<tr>
<td>Programming language</td>
<td>20.0</td>
</tr>
<tr>
<td>Programming</td>
<td>90.0</td>
</tr>
<tr>
<td>Domain</td>
<td>80.0</td>
</tr>
</tbody>
</table>

were more common among the animation group (i.e., high-level information).

Program construction

The scores for the program construction question (E-CONSTR) are presented in Figure 6. Differences between the groups are non-significant. In order to obtain a better understanding of possible differences between the groups we classified errors (excluding small syntactical errors) in several ways. In some programs, there were large omissions and it was impossible to count the number of individual errors. Thus, we could not base our analysis on error rates. Instead, we looked at whether subjects had problems in certain error types and counted the number of subjects with problems.

The first error classification considers the type of missing knowledge that would be needed to remove the error. There are three categories: programming language knowledge, i.e., problems with the syntax or semantics of Pascal, programming knowledge, e.g., problems with nested loops, and domain
knowledge, i.e., problems with understanding the goals of the program or in elementary mathematics. Table 7 contains percentages of subjects making errors of each type. Only few subjects had problems with the programming language, but almost all made programming errors and had problems in understanding the programming problem in detail. Differences between groups are statistically insignificant.

As the teaching methods were not supposed to affect domain understanding but rather programming knowledge, we analyzed programming errors in more detail. Especially, we were interested in finding at which program knowledge level subjects had problems when constructing their programs. Pennington (1987) has found that program knowledge concerning operations and control structures reflect surface knowledge, i.e., knowledge that is readily available by looking at a program. In contrast, knowledge concerning data flow and function of the program reflect deep knowledge which is an indication of a better understanding of the code. We sorted programming errors into three categories, starting from the surface level and proceeding to more complicated cases requiring deeper knowledge of the various interactions within a program:

- control: errors in the use of control structures
• **data**: errors related to data flow within and between variables

• **special data**: errors related to variable uses in exceptional conditions, e.g., sentinel input values that must not be processed

![Image](image_url)

Figure 7: Percentages of subjects making programming errors in program construction (E-CONSTR) sorted by depth of program knowledge.

Figure 7 shows that while the traditional group performed best and the animation group worst in dealing with surface structure, the opposite is true for deeper levels of program knowledge. The largest difference is between the traditional group and animation group in the special data case ($\chi^2 = 2.830, df = 1, p = 0.0998$).

We also analyzed all errors related to variables by looking at the roles of the variables and the type of error. Table 8 lists all error types sorted by the role and complexity of use. Largest differences concern complex *stepper* use with the roles group performing worst, and complex *most-recent holder* use with the animation group performing best. (All differences between groups are statistically insignificant.)

Potential origins of these errors differ. For example, misplaced initializations and updates indicate problems with control structures while the unnecessary use of an array indicates missing (or fragile (Perkins and Martin, 1986)) plan knowledge (Détienne, 2002), e.g., the subject does not know
Table 8: Errors related to variables. Numbers are percentages of subjects making the error.

<table>
<thead>
<tr>
<th>Role context</th>
<th>Group</th>
<th>Most common errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trad.</td>
<td>Roles</td>
</tr>
<tr>
<td>Simple use of <em>most-</em></td>
<td>10.0</td>
<td>23.5</td>
</tr>
<tr>
<td>recent holder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple use of <em>stepper</em></td>
<td>40.0</td>
<td>29.4</td>
</tr>
<tr>
<td>Complex use of <em>stepper</em></td>
<td>30.0</td>
<td>52.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex use of <em>most-</em></td>
<td>70.0</td>
<td>64.7</td>
</tr>
<tr>
<td>recent holder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex use of *gather-</td>
<td>70.0</td>
<td>70.6</td>
</tr>
<tr>
<td>erer*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Summary of variable related error origins in program construction (E-CONSTR).

<table>
<thead>
<tr>
<th>Variable context</th>
<th>Traditional</th>
<th>Group</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Control structures</td>
<td>Roles</td>
<td>Control structures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Domain understanding</td>
<td>Domain understanding</td>
</tr>
<tr>
<td>Complex</td>
<td>Control structures</td>
<td>Roles</td>
<td>Control structures</td>
</tr>
<tr>
<td></td>
<td>Plan knowledge</td>
<td></td>
<td>Control structures</td>
</tr>
</tbody>
</table>

how to use a *most-recent holder* to traverse through a succession of values. Similarly, a missing initialization means that the subject has used focal expansion (Rist, 1989), i.e., wrote the update before initialization, which is an indication of the lack of plan knowledge.

Finally, missing updates need more analysis because they constitute the focal line of a variable plan (Rist, 1989) and are not expected to be missed. In this program, missing updates of a simple *most-recent holder* (case (a) in Table 8) were problems with understanding the task domain, and in case (c) they were problems with control structures. In case (b), there is no evident explanation; the highest error frequency was, however, among the roles group that had obtained lowest grades in the control questions. One may wonder whether the program construction task was so demanding for their abilities that the cause for the omission was working memory overload.
Table 9 summarizes the potential error origins in simple and complex usage context. All groups had problems with control structures both in simple and in complex contexts. The animation group demonstrated best plan knowledge: they possessed required variable plans (i.e., roles) and applied them correctly. Finally, both the roles group and the animation group had problems with domain understanding—a fact we cannot explain.

**Role name usage**

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.

**Discussion**

The questions in the examination called for various programming-related activities: program simulation, program comprehension, and program construction. 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 were not easy to predict. It is therefore natural that knowledge of roles did 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 construction tasks. Our results agree with this result.

In *program comprehension*, the roles group obtained the best scores while the animation group obtained the worst scores. Surprisingly, this was true for grades of program summaries demonstrating full understanding of the program to be comprehended, also. An analysis of the program summaries revealed that the roles group gave more detailed, low-level summaries than
the animation group. Furthermore, the proportion of low-level statements in program summaries correlated positively with the grades while the proportion of high-level statements had negative correlation. Thus, teachers gave better grades for detailed answers explaining the working of the program statement by statement than for high-level summaries describing issues not directly visible in the program. As the summaries of the animation group stressed high-level information, their grades were low.

High-level information is, however, an indication of superior programming skill (Détenne, 2002; Pennington, 1987). For example, Hoadley et al. (1996) have shown that code reuse—which demonstrates expert-like programming skill—was substantially more common for students who included high-level information in their program summaries. Thus teachers in our experiment gave better grades for poorer understanding. 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.

The high-level nature of the program summaries 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 it. 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.

The analysis of the program summaries revealed also that the roles and animation groups had a better vocabulary to talk about programs. Furthermore, they gave dominantly cross-referenced program summaries whereas the traditional group produced domain-level and program-level summaries. This difference was statistically significant \( (p = 0.0046) \). Pennington (1987) used a similar method to classify program summaries and she found that high comprehension programmers almost uniformly used cross-referenced summaries while low comprehension programmers tended to produce either a program-level summary or a domain-level summary. Our results indicate that role knowledge provides students a conceptual framework that they can use to better process program knowledge.
In *program construction*, the roles and animation groups outperformed the traditional group. The scores for the roles group were better than those of the animation group but this may again be partly due to the adjusting of grades into scores which gave the greatest compensation to the roles group.

The error analysis revealed that the animation group had least problems with variables and that they had the best plan knowledge. The animation group also performed best with deep program knowledge while the traditional group performed worst. In the case of surface knowledge, the differences were opposite but small. According to Pennington (1987) deep knowledge is an indication of better understanding of the program. We may thus conclude that the traditional group had the most superficial mental model while the animation group had the best mental representation.

The differences in the level of the mental representations can be explained by role knowledge and the use of PlanAni. Role knowledge makes a difference in the position of variables within a program: in the traditional approach, a variable has no special meaning by itself but is only the object of some—to a novice apparently more or less incidental—assignments. Role knowledge turns this situation upside down: a variable is an active subject taking care of some specific task and all assignments can be seen to support this task. Thus roles make the deep program knowledge more accessible to students.

Furthermore, the PlanAni program animator makes roles easier to adopt by giving roles a visual appearance and behavior. Moreover, the visible history of previous (and sometimes even future) values stresses the succession of values of the variable. These features foster the adoption of role knowledge and may thus be assumed to increase understanding of deep properties of programs.

**Conclusions**

We have conducted a classroom experiment to study the effects of the variable roles concept and role-based animation in teaching introductory 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 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. 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.

The deeper understanding of the animator users was not, however, reflected in the course grades. Teachers gave better grades for detailed surface structure summaries 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. Moreover, this contradiction between grades and programming skills means that the evaluation of classroom experiments should not be based on simple grading but on a careful analysis of the resulting mental models.

In program simulation, the use of roles had no effect even though this may simply be a problem of grading: the form of answers to the simulation question provided no means to analyze students’ mental models. In both program comprehension and program construction, positive results were obtained. It is, however, impossible to compare the magnitude of the effect in these two cases—basically because literature on evaluating mental models in program construction is scarce as compared to that of program comprehension.

In our experiment, all experimental groups obtained basically the same teaching which was based on an existing course. Roles were introduced to the two experimental groups during the course when they first happened to occur in example programs. It would be interesting to see whether tailoring the course to better serve the role concept would increase the effects described above.

**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


