Teaching Roles of Variables in Elementary Programming Courses

Marja Kuittinen
University of Joensuu
Department of Computer Science
P.O.Box 111
FIN-80101 Joensuu, Finland
Marja.Kuittinen@cs.joensuu.fi

Jorma Sajaniemi
University of Joensuu
Department of Computer Science
P.O.Box 111
FIN-80101 Joensuu, Finland
Jorma.Sajaniemi@cs.joensuu.fi

ABSTRACT

Computer programming is a difficult skill for many students and new methods and techniques to help novices to learn programming are needed. This paper presents roles of variables as a new concept that can be used to assist in learning and gives detailed instructions on techniques to present roles to novices. These techniques are based on current learning theories and they have been used in a classroom experiment comparing traditional teaching with role-based teaching. The results suggest that the introduction of roles provides students a new conceptual framework that enables them to mentally process programs in a way similar to that of good code comprehenders; the use of role-based animation seems to assist in the adoption of role knowledge and expert-like programming skill.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.m [**Software**]: Miscellaneous—*Software psychology*

General Terms

Human factors

Keywords

roles of variables, elementary programming, teaching

1. INTRODUCTION

To learn computer programming is difficult for many students. One reason is that programs deal with abstract entities—formal looping constructs, pointers going through arrays etc.—that have little in common with everyday issues. These entities concern both the programming language in general and the way programming language constructs are assembled to produce meaningful combinations of actions in individual programs. For example, the concept of variable is very difficult for students who have several misconceptions, such as a variable could simultaneously have two values, and after an assignment statement "`A := B`", the variable `B` no longer contains a value [2, 19, 20]. Ben-Ari [2] cites unpublished study by Haberman and Ben-David Kolikant considering novices' understanding of assignment, read and write statements in Pascal. Given the statements:

```
read(A, B);
read(B);
write(A, B, B);
```

many students were not at all sure what happens when reading twice to the same variable or writing twice from the variable.

Other examples of problems on learning programming are reported by, e.g., Madison [11], Fleury [7] and Holland et al. [9]. Madison studied the internal model of parameters held by students in an introductory course. She found out that students had constructed consistent, but non-viable, models of the implementation of parameters. Likewise, Fleury noticed that students constructed their own rules for Pascal parameters that were working time to time, but not in general. Holland et al. observed that students mix up the concept of an object with other concepts like variable, class and textual representation. These findings make it clear that new methods and techniques that help students to understand these issues are needed.

So far, 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. We know only two examples of research into new concepts that can be utilized in teaching introductory programming: software design patterns, and roles of variables. Software design patterns [5] 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 [22]) and application areas (e.g., data structures [13]). Research into the use of patterns indicates that instructors should expect to refine the patterns they offer students on a regular basis [5].

Roles of variables [15, 16] 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. Ben-Ari and Sajaniemi [3] 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. In this paper, we will describe

ITiCSE'04, June 28–30, 2004, Leeds, United Kingdom.

how roles can be introduced to novices and why it should be done.

The rest of this paper is organized as follows. Section 2 gives an overview of the role concept and describes a set of ten roles that is sufficient for elementary programming courses. Section 3 gives practical advice on how to introduce and utilize roles in teaching programming. Section 4 presents the results of a classroom experiment comparing the use of roles with traditional teaching.

2. ROLES OF VARIABLES

Variables are not used in programs in a random or *ad-hoc* way but there are several standard use patterns that occur over and over again. In programming textbooks, two patterns are typically described: the counter and the temporary. The *role* of a variable [15] captures this kind of behavior by characterizing the dynamic nature of a variable: the sequence of its successive values as related to other variables and external events. The way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment.

Table 1 lists ten roles that cover 99 % of variables in novice-level procedural programs [15] and gives for each role an informal definition suitable to be used in teaching. Exact definitions of the roles can be found in the Roles of Variables Home Page [16]. For example, the exact definition of a *stepper* is as follows:

> A variable going through a succession of values depending on its own previous value and possibly on other *steppers*, *stepper followers*, and *fixed values* (e.g., a counter of input values, a variable that doubles its value every time it is updated, a variable that alternates between two values, or an index to an array that sweeps through the array using varying densities) even though the selection of possibly alternative update assignments may depend on other variables (e.g., the search index in binary search).

The informal definition in Table 1 is much shorter but yet gives the correct impression.

In this role set, an array is considered to have some role if all its elements have that role. For example, an array is a *gatherer* if it contains 12 *gatherers* to calculate the total sales of each month from daily sales given as input. Moreover, there is a special role for arrays—*organizer*.

Roles are cognitive—rather than technical—concepts. For 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 how to compute it, each new number in this sequence is a surprise. Hence, the mathematician may consider the variable as stepping through a known succession of values (i.e., a *stepper*) while the novice considers it as a *gatherer* accumulating previous values to obtain the next one.

In addition to teaching programming, roles have other applications. For example, automatic analysis may be used to find roles and thus assist maintenance engineers in program comprehension.

3. TEACHING ROLES TO NOVICES

Roles are not just a collection of additional concepts that enlarges the amount of material to be learned but they are an instrument for thinking. By using roles students are able not only to understand the life cycle of a variable but both to design and to mentally process programs in a new way. Next we describe how roles can be taught and how they can be utilized in teaching programming strategies.

Table 1: Roles of variables in novice-level procedural programming.

Role	Informal description
Fixed value	A variable initialized without any calculation and not changed thereafter.
Stepper	A variable stepping through a systematic, predictable succession of values.
Follower	A variable that gets its new value always from the old value of some other variable.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far.
Gatherer	A variable accumulating the effect of individual values.
Transformation	A variable that always gets its new value with the same calculation from values of other variables.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Temporary	A variable holding some value for a very short time only.
Organizer	An array used for rearranging its elements.

3.1 Role Knowledge Construction

According to constructivistic approach (e.g., [4, 21]) it is necessary that new knowledge is actively built on the top of existing knowledge. Constructivistic approach emphasizes that learning should be focused on understanding the essential, instead of mechanical learning by heart, and that the same issue can be interpreted and understood in different ways.

While teaching, each role can be introduced when it is encountered for the first time in some example program during lectures. It is important that the introduction of a new role is built on the top of existing information and that the distinction between the roles is explained properly.

Typically, the first program presented in a novice programming course contains literals. Then constants can be introduced by naming the literals, and the next step usually involves introducing variables. This is where the roles can be brought out. Figure 1 describes the connections that can be used as a basis for teaching. It is recommended that the *fixed value* is presented first because it can be

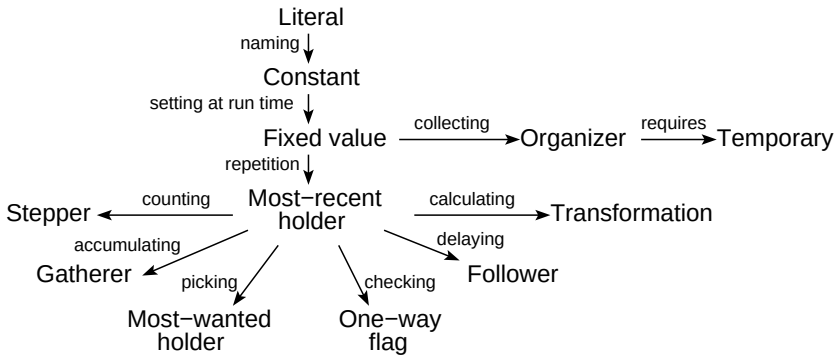

Figure 1: Relationships that can be used as a basis for incremental knowledge construction. Literal and constant are programming language constructs; other nodes are the roles.

bound to a constant by explaining that the value for such a variable can be set at run time, e.g. obtained as input, as opposed to giving the value explicitly in the program text. The next role to be introduced could be either *organizer* or *most-recent holder* but the only reasonable choice is *most-recent holder* since in introducing an *organizer* the concept of an array is needed. Having introduced the *most-recent holder*, "a repetitive fixed value", any other role having a relationship with the *most-recent holder* can be introduced.

The presentation of a role should consist of its informal definition (see Table 1) together with additional examples of its use. Moreover, the teacher may mention special cases covered by the exact definition of the role. It is essential that the distinctive features of the new role (as compared to the already known roles) are made clear.

In our lectures, we gave students a printed list describing all roles in four pages. As an example, we used the following description and short program accompanied by the role image in Figure 2(b) to introduce the *stepper* in our lectures:

> *Stepper* goes through a succession of values in some systematic way. Below is an example of a loop structure where the variable `multiplier` is used as a *stepper*. The program outputs a multiplication table while the *stepper* goes through the values from one to ten.

```
(*1*) program Multiplication_table (output);
(*2*) var multiplier: integer;
(*3*) begin
(*4*)    for multiplier := 1 to 10 do
(*5*)       writeln(multiplier, ' * 3 = ', multiplier*3)
(*6*) end.
```

> A *stepper* can also be used, for example, for counting and traversing the indexes of an array.

With this technique, lectures can be based on existing materials that need only a minor update to include the role descriptions.

3.2 Role Knowledge Consolidation

Memory elaboration involves embellishing a to-be-remembered item with meaningful additional information ([1], p. 190). Mere study of new material will not lead to better recall, but it is important how one processes the material while studying it. More meaningful processing of material results in better memories than repetition of the original information or adding non-meaningful information. For example, it is possible to repeat the informal definition of a role whenever the role reappears in some example program but this may have only a minor effect on recall. It is more important to

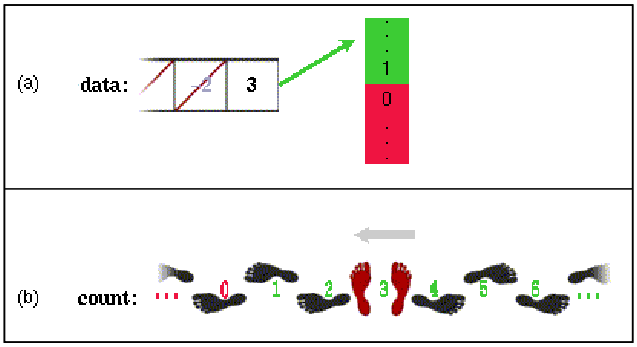

Figure 2: Visualizations of the same operation for different roles: comparing whether a *most-recent holder* (a) or a *stepper* (b) is positive.

explain how that specific variable expresses the role behavior since this is a meaningful new example of the role.

To elaborate students' memory, role names must reappear repeatedly in new meaningful contexts. For example, variable declarations may be augmented with role information, e.g.:

```
var closest: integer; (* most-wanted holder,
                         closest point to the center *)
```

These comments provide not only repetition but meaningful new examples of role behavior, e.g., a *most-wanted holder* may be a maximum value or a minimum value etc.

Another example of memory elaboration is the possibility to discuss with students about alternative role assignments. Roles are a cognitive concept which means that different people may assign different roles to the same variable. Therefore, it is meaningful to ponder how a certain variable fits the definition of a *gatherer* and at the same time the definition of a *stepper* (see the example in Section 2). Such discussions may originate from students' role assignments in their own programs or from teachers' suggestions.

Further, role images act as metaphors and provide new meaningful visual representations of roles' inherent properties. For example, the role image for a *stepper* in Figure 2(b) proposes that future values of the variable are known beforehand—an inherent property of a *stepper*. We have utilized this principle in developing a program animation system, PlanAni [17]. In addition to role images, PlanAni utilizes role information for role-based animation of operations. For example, Figure 2 gives visualizations for two syntactically similar comparisons "`some_variable > 0`". In case (a), the

variable is a *most-recent holder* and conceptually the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, allowed values with a green background and disallowed values with red. In Figure 2(b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these values are now part of the variable visualization and no new values do appear. The conceptual justification is that for a *stepper* the comparison is just a check whether the end of the known sequence of values has been reached.

When each role appears in an animation for the first time, the teacher should explain what the role image is and how it tries to visualize the most important properties of the role. Novices have little grounds to interpret visualizations in the anticipated way [12]. It is therefore possible that seeing past and future values in role images might give students the impression that all these values are available and could be accessed using some special syntax. Therefore, the teacher should stress constantly during animation sessions that only one of the values does exist in reality and that the others are shown for visualization purposes only. In our experience, this is sufficient for preventing misunderstandings.

PlanAni provides possibilities not only for mere passive viewing of ready-made animations but to engage students actively in running the animated program and in entering their own inputs. According to Hundhausen et al. [10], animation is more effective when students have an active role in using the animator.

3.3 Programming Strategies Development

Programming is a problem-solving activity requiring problem-solving operators that can be taught—preferably by giving abstract instruction and concrete examples ([1], p. 247). In our experience, novices have severe conceptual problems when they start to design a program—they may not know whether to start with a variable or a loop—indicating the lack of programming strategies, i.e., problem-solving operators. To overcome such problems, students can be taught to use roles as a starting point in program design.

As an example, consider the task of writing a program to convert temperatures between various scales (Celsius, Kelvin, Fahrenheit). Program design may start by selecting variables based on the task description. In this case, the user input consists of a temperature value and its unit. Input is normally (i.e., as taught previously in lectures) stored in a *fixed value* (if only a single value is read) or a *most-recent holder*; so one of those is needed for both inputs; let's call them `degree` and `unit`. The variable declaration does not depend on the role; the difference is that a *most-recent holder* induces the use of a loop because in every program seen in lectures a *most-recent holder* has occurred in the context of a loop. Let us assume that only a single temperature will be input, so *fixed values* will suffice.

The program has to produce the same temperature in several units. These are *transformations* of the original value, so we will need one for each of them. The program looks now as follows:

```
program ?name? (input, output);
var degree:        ?type? (* fixed value: input temperature *)
    unit:          ?type? (* fixed value: input unit        *)
    degCelsius:    ?type? (* transformation: input in C     *)
    degKelvin:     ?type? (* transformation: input in K     *)
    degFahrenheit: ?type? (* transformation: input in F     *)

begin
    input degree
    input unit
    degCelsius := ?calculation based on input degree?
    degKelvin := ?calculation based on input degree?
    degFahrenheit := ?calculation based on input degree?
    output all degrees
end.
```

This program uses two roles and their prototypical uses in programs. It does not say what should be written between question marks but it gives some hints on the kind of things there might be.

There is no guarantee that this method produces an optimal program but it gives a possibility to start program design directly on the basis of the problem statement. Thus, role information can be used to teach elementary programming strategies.

4. AN EXPERIMENT ON USING ROLES IN TEACHING

We have conducted a classroom experiment during an introductory Pascal programming course using teaching methods described in the previous section. A more thorough description of the experiment and its results can be found elsewhere [18]; here we only summarize the main conclusions.

The subjects of the experiment—ninety-one Finnish undergraduate students studying computer science for the first semester—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 (*the traditional group*); using roles throughout the course (*the roles group*); and using roles together with the use of the animator in exercises (*the animation group*). The course lasted five weeks, with four hours of lectures and two hours of exercises each week. During exercises, all groups animated four programs. For animation, the animation group used PlanAni and the other groups used a visual debugger (Turbo Pascal v. 7.0) with all variables added to the watch panel of the debugger. 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.

Subjects attended an examination after the course. Their answers were graded for the purposes of the course but, in addition, they were further analyzed for the purposes of the experiment. One of the tasks in the examination was to write a summary of a given short program. These program summaries were analyzed using Good's program summary analysis scheme [8] based on the level of expressions, e.g., whether objects were referenced in program terms ("variable `w`") or in domain terms ("patient's weight"). 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 more evenly.

To analyze this difference further, we used a similar strategy as Pennington [14] and sorted program summaries into three types depending on the amount of domain vs. program statements in object descriptions. Summaries with at least 67 % domain statements were called *domain-level summaries*, summaries with at least 67 % program statements were classified as *program-level summaries*, and all others were called *cross-referenced summaries* because they had a more even distribution of domain and program information. 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. Our result thus indicates 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.

We also analyzed the types of errors made in another task: writ-

ing a new program. We were especially interested in finding at which *program knowledge level* subjects had problems when constructing their programs. According to Pennington [14], 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. While the traditional group performed best and the animation group worst in dealing with surface structure, the opposite was true for deeper levels of program knowledge. Thus, the teaching of roles seems to assist in the adoption of programming strategies related to deep program structures, i.e., use of variables.

Furthermore, in the program summary task, animation users—when compared to the roles group—tended to stress deep program structures as opposed to directly visible operations and control structures. Previous research has shown that summarizing programs using deep structures is an indication of superior programming skill [6, 14]. For example, Clancy and Linn [5] cite a study demonstrating that code reuse—which demonstrates expert-like programming skill—was substantially more common for students who gave data, i.e., deep, level program summaries. Thus, the use of PlanAni in exercises seems to assist in the adoption of expert-like programming skill.

5. CONCLUSION

Learning to write computer programs is a hard task for many students and research into techniques to help them is needed. We have suggested that the role concept may be used to assist in learning and we have given detailed instructions on how to present roles to novices and how to utilize this knowledge in program design. We have also reported an experiment comparing traditional teaching with role-based teaching and animation.

The results suggest that the introduction of roles provides students a new conceptual framework that enables them to mentally process program information in a way similar to that of good code comprehenders. The use of role-based animation seemed to assist in the adoption of role knowledge and expert-like programming skill. The changes in instruction when adding roles to traditional teaching were minimal. Based on the results we recommend the introduction of the roles in elementary programming courses.

6. REFERENCES

[1] J. R. Anderson. *Cognitive Psychology and Its Implications*. Worth Publishers, 5th edition, 2000.

[2] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[3] M. Ben-Ari and J. Sajaniemi. Roles of variables as seen by CS educators. In *9th Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'04)*, 2004.

[4] C. Bereiter. Constructivism, socioculturalism and Popper's world. *Educational Researcher*, 23(7):21–23, 1994.

[5] M. J. Clancy and M. C. Linn. Patterns and pedagogy. In *Proc. of the 30th SIGCSE Technical Symposium on CS Education*, volume 31(1) of *ACM SIGCSE Bulletin*, pages 37–42, 1999.

[6] F. Détienne. *Software Design—Cognitive Aspects*. Springer-Verlag, 2002.

[7] A. E. Fleury. Parameter passing: The rules the students construct. In *Proc. of the 22nd SIGCSE Technical Symposium on CS Education*, volume 23(1) of *ACM SIGCSE Bulletin*, pages 283–286, 1991.

[8] J. Good and P. Brna. Toward authentic measures of program comprehension. In *EASE and PPIG 2003, Papers from the Joint Conference at Keele University*, pages 29–49, 2003.

[9] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. In *Proc. of the 28th SIGCSE Technical Symposium on CS Education*, volume 29(1) of *ACM SIGCSE Bulletin*, pages 131–134, 1997.

[10] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002.

[11] S. K. Madison. *A Study of College Students' Construct of Parameter Passing: Implications for Instruction*. PhD thesis, University of Wisconsin, 1995.

[12] P. Mulholland and M. Eisenstadt. Using software to teach programming: Past, present and future. In J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization – Programming as a Multimedia Experience*, pages 399–408. The MIT Press, 1998.

[13] D. Nguyen. Design patterns for data structures. In *Proc. of the 29th SIGCSE Technical Symposium on CS Education*, volume 30(1) of *ACM SIGCSE Bulletin*, pages 336–340, 1998.

[14] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Ablex Publishing Company, 1987.

[15] J. Sajaniemi. 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)*, pages 37–39. IEEE Computer Society, 2002.

[16] J. Sajaniemi. Roles of variables home page. http://www.cs.joensuu.fi/~saja/var_roles/, 2003. (Accessed Nov. 12th, 2003).

[17] J. Sajaniemi and M. Kuittinen. Program animation based on the roles of variables. In *Proc. of ACM 2003 Symposium on Software Visualization (SoftVis 2003)*, pages 7–16. Association for Computing Machinery, 2003.

[18] J. Sajaniemi and M. Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, In press.

[19] R. Samurçay. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 161–178. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[20] D. Sleeman, R. T. Putnam, J. A. Baxter, and L. Kuspa. A summary of misconceptions of high school Basic programmers. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 301–314. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[21] E. von Glasersfeld. A constructivist approach to teaching. In P. Steffe and J. Gale, editors, *Constructivism in Education*, pages 3–15. Lawrence Erlbaum Associates, Hillsdale (NJ), 1995.

[22] E. Wallingford. The elementary patterns home page. http://www.cs.uni.edu/~wallingf/patterns/elementary/, 2003. (Accessed Nov. 12th, 2003).