HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 62–74

A ROLES-BASED APPROACH TO VARIABLE-ORIENTED PROGRAMMING

Juha Sorva
Helsinki University of Technology
Finland

Abstract: *Delocalized variable plans pose problems for novice programmers trying to read and write programs. Variable-oriented programming is a programming paradigm that emphasizes the importance of variable-related plans, and localizes actions pertaining to each variable together in one place in the program code. This paper revisits the idea of variable-oriented programming and shows how it can be founded on roles of variables: stereotypes of variable use suitable for teaching to novices. The paper sketches out how variable-oriented, roles-based programming could be implemented using either a new programming language or a framework built on an existing language. The possible applications, merits, and problems of a roles-based approach, and variable-oriented programming in general, are discussed. This paper points toward possible research directions for the future and provides a basis for further discussions of variable-oriented, roles-based programming.*

Keywords: *roles-based programming, variable-oriented programming, roles of variables, delocalized plans, programming languages.*

INTRODUCTION

It has been widely noted that novice programmers have great difficulty in comprehending and creating computer programs (for recent reports, see Lister et al., 2004; McCracken et al., 2001). A partial explanation for this is provided by the novices' lack of programming-related schemas or plans (Détienne, 1990; Soloway & Ehrlich, 1986). *Schemas* are mental knowledge structures for storing abstract information that can be applied when planning solutions to specific problems that fall within the scope of the schema. An expert in a domain possesses a wide array of rich, domain-specific schemas that reduce cognitive load during problem-solving tasks, such as programming and enable solving more complex problems. An expert's problem-solving process is characterized by planning ahead and forward development (Byckling & Sajaniemi, 2006a; Rist, 1989).

Many schemas in programming are related to the use of variables (Soloway, Ehrlich, Bonar, & Greenspan, 1982). For instance, a basic programming schema could describe how variables can serve as "counters," whose values start at zero and are then repeatedly incremented

DOI: http://dx.doi.org/10.17011/ht/urn.200804151353

by one. Commonly, the ways in which a variable is used in a program are not defined by a single line of code or even by consecutive lines; references to each variable are spread throughout the program code. In the terminology of Soloway, Lampert, Letovsky, Littman, and Pinto (1988), the plan for such a variable is *delocalized*. Delocalization of a plan increases the cognitive load of a programmer trying to comprehend it, since multiple separate units have to be kept in working memory at once in order to figure out the plan. Novice programmers may find coping with this cognitive load very difficult. Delocalized plans can be clarified with documentation (Soloway et al., 1988) or software tools (Sajaniemi & Niemeläinen, 1989). In recent years, *roles of variables* have been introduced as a means to describe, discuss, and think about common stereotypes of variable usage (Sajaniemi, 2002, 2003). Roles of variables have been used to document variable plans and for other purposes in teaching introductory programming (Byckling & Sajaniemi, 2007; Sajaniemi & Kuittinen, 2005; Sorva, Karavirta, & Korhonen, 2007).

This paper presents ongoing work on *variable-oriented programming*, a programming paradigm that places an emphasis on localizing variable-related actions in program code. This work draws on prior work on roles of variables, and uses roles as a basis for creating variable-oriented programs. The paper is structured as follows. The Related Work section describes previous work on roles of variables and variable-oriented programming. The A Roles-Based Approach section introduces a new approach to variable-oriented programming, and discusses how it could be implemented, using either a custom-made programming language or existing programming languages. The Discussion section then takes a look at the possible uses, merits, and downsides of the new approach. The paper concludes with general comments and a look at possible future work.

RELATED WORK

Roles of Variables

Roles of variables are stereotypes of variable use in computer programs (Sajaniemi, 2002). Roles embody expert programmers' tacit knowledge of variable usage patterns, which can be made explicit and taught to students (Sajaniemi & Navarro Prieto, 2005). Roles can help teachers explain delocalized variable-related schemas in programs and assist in the stepwise refinement of pseudocode designs of algorithms (Sorva et al., 2007). Prior research suggests that introductory-level students who are taught programming using roles of variables gain better program comprehension skills than students taught in an otherwise similar way but without using roles (Sajaniemi & Kuittinen, 2005). Moreover, roles-based instruction facilitates the development of program construction skills better than traditional instruction, especially if roles-based visualizations of programs are also used in teaching (Byckling & Sajaniemi, 2006b, 2007).

According to Sajaniemi's (2002) research, the behavior of 99% of variables in novice-level programs can be characterized within a small set of roles. The following list, reprinted from Sorva et al. (2007, p. 410), briefly introduces each variable role. For a fuller introduction to roles of variables, and concrete program examples of each role, see Sajaniemi (2003).

1. A variable has the role *fixed value* if the variable's value is not changed after it is initialized.

2. A variable has the role of *stepper* if it is assigned values in a systematic and predictable order. An example of a stepper is an index counter used when looping through an array of elements.
3. A variable has the role of *most-recent holder* if it holds the latest value in a sequence of unpredictable data values. For instance, a most-recent holder could be used to store the latest element encountered while iterating through a collection of data elements, or the latest value that has been assigned to an object's attribute (i.e., to an instance variable that is a most-recent holder) by a setter method.
4. The role *most-wanted holder* describes variables that hold the "best" value encountered in a sequence of values. Depending on the program and the type of the data, the best value may be the largest, smallest, alphabetically first, or an otherwise most appropriate value.
5. A variable has the role *gatherer* if the variable is used to somehow combine data values that are encountered in a sequence of values, and the variable's value represents this accumulated result. For instance, a variable keeping track of the balance of a bank account (e.g., the sum of deposits and withdrawals) is a gatherer.
6. A *follower* is a variable that always holds the most recent previous value of another variable. Whenever the value of the followed variable changes, the value of the follower is also changed. For example, the "previous node pointer" used when traversing a linked list is a follower.
7. A variable is a *one-way flag* if it only has two possible values and if a change to the variable's value is permanent. That is, once a one-way flag is changed from its initial value to the other possible value, it is never changed back. For example, a Boolean variable keeping track of whether or not errors have occurred during processing of input is a one-way flag.
8. A variable has the role *temporary* if the value of the variable is needed only for a short period. For example, an intermediate result of a calculation can be stored in a temporary in order to make it more convenient or efficient to use in later calculations.
9. An *organizer* is a variable that stores a collection of elements for the purpose of having that collection's contents rearranged. An example of an organizer is a variable that contains an array of numbers during sorting.
10. A variable is a *container* if it stores a collection of elements in which more elements can be added (and, typically, can be removed as well). For example, a variable that references a stack could be a container.
11. A *walker* is a variable whose values traverse a data structure, moving from one location in the structure to another. For instance, a variable that contains a reference to a node in a tree traversal algorithm and a variable that keeps track of the search index in a binary search algorithm can be considered to be walkers.

Variable-Oriented Programming

In traditional procedural and object-oriented programming, the behavior of a variable, that is, the logic that dictates how the variable is used, is often defined at multiple distinct locations in program code. Depending on the scope of the variable, the behavior may be described by

inconsecutive lines of code within a function or method, may be located in a number of functions, or even located in several program modules. Declaring a variable, if it is explicitly done in the language at all, is a matter separate from the variable's behavior.

There is an alternative way to organize variable behavior in programs. If a variable's behavior pattern is defined at the variable's declaration, the "usage plan" of the variable becomes localized in one place. This idea is central to the variable-oriented way of programming discussed in this paper. In a *variable-oriented program*, each variable declaration is accompanied by a definition of how the variable's value is initialized and later updated. A variable declaration could also include information of when the variable's value is read and dependencies on other variables. In a variable-oriented program, such rich variable declarations serve as the basis of, and indeed govern, the creation of algorithms.

Variable-oriented programming has made an appearance in literature before. It was introduced in connection with the program editor VOPE, which makes use of variable-orientation to provide multiple views of program code written in the Pascal language (Sajaniemi & Niemeläinen, 1989). In addition to a traditional control-flow oriented view of Pascal programs, VOPE shows a purely variable-oriented view, which groups code fragments so that all references to each variable are gathered together.

A ROLES-BASED APPROACH

A look at how an algorithm could be devised using roles of variables may be useful. The passage below presents a hypothetical thought pattern of how a student of programming, who has been taught to use the roles of variables, might go about the task of creating an algorithm for computing the *n*th Fibonacci number.

> Some way of keeping track of consecutive Fibonacci numbers is needed to compute to the *n*th one. Each new value is produced by computing a new value based on the current one. That's a job for a gatherer. And since, in this case, each new value is computed based on two older values, a follower is needed to store the older value of the gatherer. By starting from the first Fibonacci number (one), then after n-1 updates to the gatherer, the result should be reached.

While fictional and idealized, this example offers an idea of how roles-based reasoning might proceed and make use of the common patterns of variable use embodied by roles of variables. It is also an example of thinking ahead: The programmer uses existing schemas to plan in advance how he/she will use the two variables. Figure 1 shows a somewhat more formal and complete description of the algorithm, using a pseudocode notation that closely reflects the reasoning process described above.

In the pseudocode in Figure 1, two variables are declared, each with a different role. For each variable, its behavior has been declared as a part of the variable definition. The example illustrates how an algorithm can be built by attaching behavior to variable definitions. Further, it shows how roles of variables can serve as templates for common patterns in a variable-oriented program.

Each variable is declared as an instance of a role, which determines the kinds of operations that need to be defined for each instance of the role. For example, all gatherers require a definition of how their values change as a function of the same variable's old value,

```
define GATHERER curr:
        initial value is 1
        always updated by computing value of curr + prev

define FOLLOWER prev:
        initial value is 0
        follows curr (and always receives its old value)

make n-1 updates to curr (results in changes to both curr and prev)
print curr (which now holds the nth Fibonacci number)
```

Figure 1. Variable-oriented pseudocode.

whereas a follower is dependent on another variable whose old values it receives. For a fixed value (not shown in the example), only an initialization is needed, while a most-wanted holder would define an operation to test whether a given value is "more wanted" than the current value, and so on.

The next two subsections explore possible implementations for variable-oriented, roles-based algorithms such as that in Figure 1. The first one sketches out a variable-oriented programming language that uses roles of variables as language-level abstractions. The second then takes a look at how a similar framework could be implemented in an existing programming language.

A Roles-Based Language

Figure 2 provides an example of variable-oriented code based on roles of variables. It is written in a speculative language called ROTFL (Role-Oriented, Titillating but Fictional Language). The reader should note that ROTFL is at a draft stage and lacks a full syntactical and semantical specification. The notation is used here to provide "food for thought." In Figure 2 and in other Fibonacci examples in this paper, *n* is an integer-valued constant that determines which Fibonacci number is to be printed out.

```
Gatherer curr:
   inits to: 1
   updates with: curr + prev

Follower prev:
   inits to: 0
   follows: curr

update curr times n-1
print(curr)
```

Figure 2. The Fibonacci algorithm in the language ROTFL.

In ROTFL, there are no traditional variable definitions. Instead, all variables are defined in terms of roles and associated with behaviors appropriate for those roles. Roles of variables are language-level constructs, and there are reserved words related to defining or using variables with particular roles (e.g., follower, update). ROTFL does not feature assignment operators or statements in the traditional sense. Instead, variables' values are changed in role-specific ways. For instance, values are assigned to gatherers with the reserved word *update*, which uses the updates-with operation of the gatherer to compute a new value for the variable, and followers receive new values implicitly as the value of a followed variable changes.

Traditional loops are also conspicuous by their absence in Figure 2, despite the fact that the algorithm is an iterative one. In this example, repetition is achieved using the keyword *times* in association with updating the value of the gatherer curr. Another mechanism for achieving repetition is illustrated in Figure 3, where a *do each* command repeatedly updates a most-recent holder variable until a condition associated with the variable is reached. The same example also shows a most-wanted holder dependent on a most-recent holder that serves as its *source*.

```
MostRecentHolder input:
   updates with: readLine()
   until: input == 'stop'

MostWantedHolder longestInput:
   source: input
   wants value if: value.length() > longestInput.length()

do each input
print(longestInput)
```

Figure 3. A ROTFL code fragment to read in lines and print out the longest one.

Implementing Roles in an Existing Language

Variable-oriented programming can also be done within an existing programming language, provided a suitable framework is available. Figure 4 shows how the variable-oriented, roles-based program from Figures 1 and 2 can be written in the Python language. The program makes use of an anonymous function defined using Python's lambda mechanism.

The program in Figure 4 relies on a framework that defines roles of variables as Python classes, and role-related operations (such as updating the value of a gatherer) as methods of these classes. A partial framework for this purpose, defining the classes gatherer and follower, is given in the Appendix.

```
curr = Gatherer(1, lambda: curr + prev)
prev = Follower(0, curr)
curr.updateTimes(n-1)
print(curr)
```

Figure 4. A variable-oriented code fragment in Python.

DISCUSSION

Uses of Roles-Based Programming

As noted in the introduction to this paper, prior research suggests that the behavior of 99% of variables can be characterized with a small set of roles, at least within novice-level programs (Sajaniemi, 2002). It does not immediately follow, however, that 99% of even novice-level programs can be conveniently written as variable-oriented programs using roles as templates for variable behavior. Nevertheless, it seems roles form a solid foundation for creating variable-oriented programs, as the small role set provides a quite substantial number of variables with templates that capture some key aspects about how those variables are used. This matter calls for further study.

Variable-oriented programming localizes variable plans in program code. Prior work in cognitive psychology of programming suggests that it is likely that localizing variable plans facilitates the extraction and construction of variable-related schemas (Soloway et al., 1988) and therefore aids novices in acquiring some key programming skills. With this in mind, and in light of previous experiences of using roles of variables in teaching, one can speculate whether a variable-oriented, roles-based language could be useful for teaching introductory programming. Clearly, there could be merits to such an approach if variable-orientation helps students construct variable-related schemas, if roles can be used to encourage forward development (Byckling and Sajaniemi, 2007), and if there were roles-aware program development tools that could provide helpful feedback and error messages.

There also clearly are problems with such an approach. Not least of these is that while variable-oriented programs emphasize variable-related plans and the data flow of programs, the control flow of the program is not in focus. Understanding "what happens when" during the execution of a variable-oriented program may be quite difficult, especially for the beginner. There is a trade-off between emphasizing variable-related schemas and emphasizing control-flow-related schemas. Using tools similar to VOPE (Sajaniemi & Niemeläinen, 1989), which provides multiple views of programs, could be useful in combining these different aspects of programs. A notation based on roles of variables could be used to build variable-oriented views and to link them to procedural or object-oriented views.

Depending on the notation used, a variable-oriented program can be quite self-documenting of variable-related schemas (see, e.g., Figure 2). Roles of variables help in this, since role names succinctly describe patterns of variable use. However, it is not immediately obvious what the documentative value of variable-oriented notations is compared to non-variable-oriented notations that explicate the role of each variable (e.g., by simply tagging each variable declaration with a role name using code comments). Documenting delocalized variable behavior using role names may often do enough and using a variable-oriented language may be overkill for this purpose.

Even if beginners are not taught variable-oriented, roles-based programming directly, they might indirectly benefit from it. Bergin (2005) suggests that instructors of programming (and others) could benefit from "etudes" that take one particular programming technique to an extreme. While such etudes have no intrinsic value of their own, they can help hone one's skills in a particular technique and to ingrain that technique into one's thinking. For helping instructors (not novice programmers) make use of polymorphism, Bergin suggests the following etude:

> Find some old program that you have around and that you are proud of…. Strictly as an etude, rewrite that program with NO if/switch statements: no selection at all. Solve all of the problems your ifs solve with polymorphism. (Bergin, 2005, p. 1)

In a similar vein, roles-based programming could serve as an etude for using roles of variables in general. The intellectual exercise of rewriting programs in a variable-oriented way, using roles as templates for variables, with no traditional-style assignment and perhaps with no traditional-style loops, could deepen instructors' understanding of roles and help them think of algorithms in terms of variables and roles. At least, the exercise has expanded the mind of this author.

Variable-Oriented "Purity"

According to Sajaniemi and Niemeläinen (1989, p. 67, my emphases), "Variable-oriented programming is a new programming paradigm which collects *all* actions concerning any single variable together.... The plan of a variable is clearly visible and *totally* described in the variable definition."

A "pure" variable-oriented program, then, would gather all references (assignments and reads) to a variable into one complete variable definition, irrespective of the location of these references in the control flow of the program. The reader may note that the examples shown in this paper are not pure by this strict definition. For instance, in Figure 2, neither the command *update* nor reading the variable's value for printing purposes (i.e., the last two lines) is located within the variable definition. The example can be seen as a hybrid that is largely variable-oriented but partially control-flow-oriented. It can be contrasted with the pure variable-oriented views displayed by the VOPE tool (Sajaniemi & Niemeläinen, 1989).

Roles of variables are concerned with assignment, with change (or lack of change) in the values of variables, and with the way consecutive values of variables are related to each other. Roles are not concerned with *when* a variable's value is updated or read, or with what is done with the value after it has been read (whether it is printed, passed as a parameter, or something else). A variable-oriented program based solely on roles of variables will not be pure. A more complete discussion of the purity of variable-orientation is beyond the scope of this paper. The next subsection also touches on the issue of purity, however, as it briefly explores the relationship between object-oriented programming, variable-orientation, and roles-based programming.

Compatibility with Object-Orientation

The original set of roles of variables was discovered by analyzing procedural programs. Since then, roles of variables have been applied to object-oriented as well as functional programs (Sajaniemi, Ben-Ari, Byckling, Gerdt, & Kulikova, 2006). Roles seem to be a useful tool irrespective of the programming paradigm used.

What, then, is the relationship between variable-orientation and object-orientation? Quoting again from Sajaniemi and Niemeläinen (1989, p. 67), "In object-oriented programming all operations applicable to objects of a class are described in one place.... In variable-oriented programming programs center around the variables. A variable, and all the actions using that particular variable, are described in one place."

One of the two paradigms elevates classes as a key abstraction around which program code is structured; the other does the same to variables. These two abstractions are in competition, but not incompatible. It is quite possible to envision a hybrid of the object-oriented and variable-oriented paradigms, as illustrated by the example in Figure 5.

It is easy to see that Figure 5 is not pure in terms of variable-orientation. The generic plan for using the instance variable *balance*, a gatherer, is defined at the variable declaration. However, the precise ways in which the three methods make use of this generic plan are spread out in the code.

```
class Account:
   private Gatherer balance:
      inits to: 0
      updates with (FixedValue amount):
         if (balance + amount < 0) then:
            0
         else:
            balance + amount

      public method deposit(FixedValue depositSize):
         update(depositSize) balance

      public method withdraw(FixedValue withdrawalSize):
         update(-withdrawalSize) balance

      public method getBalance():
         balance
```

Figure 5. A ROTFL class representing simple bank accounts with non-negative balances.

Another issue needs to be considered when applying roles of variables to object-oriented programs. As was noted by Sorva et al. (2007, p. 419),

> Annotating a member variable and a local variable with the same role name indicates that we think of them as similar. However, our experience suggests that in many people's perception a most-recent holder member variable, for instance, is used rather differently than a most-recent holder local variable. A settable attribute of an object (the name of a person object, say) is experienced as being quite different from a local variable that stores the most recent element encountered in a collection during iteration.... This kind of dividedness of roles is potentially confusing….

It may be that, in order to apply roles-based programming to object-oriented programs, new roles are needed to represent different uses of instance variables. As an example, a role name *settable attribute* could better describe the purpose of most-recent holder instance variables. If needed, the roles-based language or framework could provide a somewhat different template for settable attributes than for other most-recent holders.

CONCLUSIONS AND FUTURE WORK

In this paper, I have revisited the previously discovered ideas of variable-oriented programming and roles of variables. This paper combines these two ideas by founding variable-orientated programs on roles, and sketches out how such a roles-based approach could be implemented using a roles-based programming language or a framework written in another language. The paper has described ongoing work on tools for roles-based programming, and discussed the possible applications, merits, and problems of the approach. It is my hope that this paper can serve as a basis for further discussions of variable-oriented, roles-based programming.

This paper has merely introduced the idea of using roles of variables in variable-oriented programming. There are many research paths that could be followed in the future. Roles-based languages or frameworks could be developed further from the drafts presented, investigating the suitability of the variable-oriented approach for more complex programs. Ways of defining dependencies between variables could be explored, as could the idea of actions that trigger when variables' values change. Here, inspiration could perhaps be drawn from earlier work, such as the language EDEN (Yung, Joy, & Ward, 1987), which, although not variable-oriented, allows the programmer to associate "action specifications" to variables.

The suitability of the current set of roles of variables for roles-based programming needs exploring, as does the idea of custom roles defined by the programmer. The possible usefulness of roles-based programming outside educational settings could be investigated.

The effects of a variable-oriented notation on understanding programs' control flow will need to be explored if this approach is to be taken further. Roles-based tools supporting both variable-oriented and other views of programs could be developed. If the approach looks promising, the potential of variable-oriented programming in instruction could be evaluated.

Using roles-based programming as an etude for instructors to deepen their understanding of roles of variables seems like a promising avenue to take in the future. This can be done even using a speculative language like ROTFL.

REFERENCES

Bergin, J. (2005, July). *Variations on a polymorphic theme: An etude for computer programming*. Paper presented at the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, Glasgow, UK. Retrieved April 15, 2007, from http://www.cs.umu.se/~jubo/Meetings/ECOOP05/Submissions/Bergin-full.pdf

Byckling, P., & Sajaniemi, J. (2006a). A role-based analysis model for the evaluation of novices' programming knowledge development. In *ICER '06: Proceedings of the 2006 International Workshop on Computing Education Research* (pp. 85–96). New York: ACM Press.

Byckling, P., & Sajaniemi, J. (2006b). Roles of variables and programming skills improvement. *SIGCSE Bulletin*, *38*, 413–417.

Byckling, P., & Sajaniemi, J. (2007). A study on applying roles of variables in introductory programming. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC '07; pp. 61–68). Coeur d'Alène, ID, USA: IEEE Computer Society.

Détienne, F. (1990). Expert programming knowledge: A schema-based approach. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 205–222). London: Academic Press.

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., & Sanders, K. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, *36*, 119–150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, *33*, 125–180.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science, 13*, 389–414.

Sajaniemi, J. (2002). 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* (pp. 37–39). Arlington, VA, USA: IEEE Computer Society.

Sajaniemi, J. (2003). The roles of variables home page. Retrieved April 15, 2007, from http://cs.joensuu.fi/~saja/var_roles

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education*, *16*, 261–279.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, *15*, 59–82.

Sajaniemi, J., & Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group* (PPIG; pp. 145–159). Brighton, UK: University of Sussex.

Sajaniemi, J., & Niemeläinen, A. (1989). Program editing based on variable plans: A cognitive approach to program manipulation. In *Proceedings of the Third International Conference on Human-computer Interaction on Designing and Using Human-computer Interfaces and Knowledge Based Systems* (2nd ed.; pp. 66–73). New York: Elsevier Science Inc.

Soloway, E., & Ehrlich, K. (1986). Empirical studies of programming knowledge. In C. Rich & R. C. Waters (Eds.), *Readings in artificial intelligence and software engineering* (pp. 507–521). San Francisco: Morgan Kaufmann Publishers Inc.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), *Directions in human-computer interactions* (pp. 27–54). Norwood, NJ, USA: Ablex Publishing.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., & Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM*, *31*, 1259–1267.

Sorva J., Karavirta V., & Korhonen A. (2007). Roles of variables in teaching. *Journal of Information Technology Education*, *6*, 407–423.

Yung, E., Joy, M., & Ward, A. (1987). *EDEN: The engine for definitive notations*. Retrieved April 15, 2007, from http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/

Author's Note

All correspondence should be addressed to:
Juha Sorva
Helsinki University of Technology
Department of Computer Science and Engineering
Konemiehentie 2
02015 TKK, Finland
jsorva@cs.hut.fi

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

APPENDIX

A PARTIAL FRAMEWORK FOR VARIABLE-ORIENTED, ROLES-BASED PROGRAMMING IN PYTHON

The classes below form a partial (but working) framework for writing variable-oriented programs in terms of roles of variables in the Python language. The partial framework shown here has implementations for only some main features of three roles (fixed value, gatherer and follower). For an example of using the classes, see Figure 4.

Other variable roles can be implemented in Python along the same lines. Implementation-wise, most-recent holders are simple; they just need an update method that replaces the old value with the given new one. Steppers and most-wanted holders can be implemented similarly to gatherers and most-recent holders, respectively. Temporary variables are akin to fixed values and trivial to implement, one-way flags likewise. Containers need a more complex class, with methods for adding and removing values. Alternatively, containers could be left unimplemented as an explicit role, relying on Python's built-in data structures instead. Organizers are characterized by a variable-specific function that defines a means for ordering data, which can be passed as a constructor parameter (cf. the gatherer implementation below).

The few variables that do not have any of the roles in Sajaniemi's (2003) role set can be treated as most-recent holders, or with a functionally similar but differently named class (e.g., *special*), which effectively allows new values to be assigned freely to the variable by passing them as a parameter to update. Alternatively, programmers could define their own program-specific custom roles.

```
import types

class Role:
    def __init__(self, initsTo):
        self.followers = []
        if (type(initsTo) == types.FunctionType):
            self.value = initsTo()
        else:
            self.value = initsTo

    def __add__(self, x):
        return self.value + x
    __radd__ = __add__

    def __str__(self):
        return repr(self.value)

    def addFollower(self, follower):
        self.followers.append(follower)
```

```
class FixedValue(Role):
    def __init__(self, initsTo):
        Role.__init__(self, initsTo)


class Gatherer(Role):
    def __init__(self, initsTo, updatesWith):
        Role.__init__(self, initsTo)
        self.updatesWith = updatesWith

    def update(self):
        oldValue = self.value
        self.value = self.updatesWith()
        for f in self.followers:
            f.update(oldValue)

    def updateTimes(self, times):
        for time in range(times):
            self.update();


class Follower(Role):
    def __init__(self, initsTo, followedVariable):
        Role.__init__(self, initsTo)
        followedVariable.addFollower(self)

    def update(self, newValue):
        oldValue = self.value
        self.value = newValue
        for f in self.followers:
            f.update(oldValue)
```