A Roles-Based Approach to Variable-Oriented Programming

Juha Sorva

Helsinki University of Technology
jsorva@cs.hut.fi

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 schemas and brings actions pertaining to each variable together in one place in 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 at possible research directions for the future and provides a basis for further discussions of variable-oriented, roles-based programming.

1 Introduction

It has been widely noted that novice programmers have great difficulty in comprehending and creating computer programs (see [1, 2] for recent reports). A partial explanation for this is provided by novices' lack of programming-related *schemas* or *plans* [3, 4]. 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 the solving of more complex problems. An expert's problem-solving process is characterized by planning ahead and forward development [5, 6].

Many schemas in programming are related to the use of variables [7]. For instance, a basic programming schema could describe how variables can serve as 'counters' whose values start at zero and are then repeatedly incremented 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 out in the program code. In the terminology of Soloway et al., the plan for such a variable is *delocalized* [8]. 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.

Delocalized plans can be clarified with documentation [8] or software tools [9]. In recent years, *roles of variables* have been introduced as a means to describe, discuss and think about common stereotypes of variable usage [10, 11]. Roles of variables have been used to document delocalized variable plans and for other purposes in teaching introductory programming [12, 13].

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. Section 2 introduces related work on roles of variables and variable-oriented programming. Section 3 describes a new roles-based approach to variable-oriented programming, and discusses how it could be implemented either using a custom-made programming language or existing programming languages. The possible uses, merits and downsides of the approach are examined in Section 4. Section 5 takes a look at possible future work and Section 6 provides a short conclusion.

2 Related Work

2.1 Roles of Variables

Roles of variables are stereotypes of variable use in computer programs [10]. Roles embody expert programmers' tacit knowledge on variable usage patterns, which can be made explicit and taught to students [14]. Roles can help teachers explain delocalized variable-related schemas in programs and assist in the stepwise refinement of pseudocode designs of algorithms [13]. 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 [12], and that roles-based instruction also facilitates the development of program construction skills better than traditional instruction especially if roles-based visualizations of programs are also used in teaching [15].

The following list, replicated from [13], briefly introduces each variable role. For a more verbose introduction to roles of variables, and concrete program examples of each role, see [11].

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 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 array elements.
3. A variable has the role 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 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 object (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.

2.2 Variable-Oriented Programming

In traditional procedural and object-oriented programming, the behavior of a variable, i.e., the logic that dictates how the variable is used, is often defined by multiple distinct locations in program code. Depending on the scope of the variable, the behavior may be described by unconsecutive lines of code within a function or method, or may be located in a number of functions or even 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 with other variables. In a variable-oriented program, such rich variable declarations serve as the basis for, and indeed drive, 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 to program code written in the Pascal language [9]. 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.

The next section explores how roles of variables could be used as a basis for variable-oriented programming and shows concrete examples of variable-oriented programs.

3 A Roles-Based Approach

Let us take a look at how an algorithm could be devised using roles of variables. Below are shown the musings of a student of programming, who has been taught using roles of variables, and is faced with the task of creating an algorithm for computing the nth Fibonacci number.

> I need some way of keeping track of consecutive Fibonacci numbers that I compute to reach the `n`th one. That's a job for a GATHERER, I guess, since a GATHERER gets its values by computing a new accumulated value based on the current one. Oh, and since in this case each new value is computed based on *two* older values, I'll also need a FOLLOWER to store the older value of the GATHERER. If I start from the first Fibonacci number (one), then after `n-1` updates to the GATHERER, I'll have the result.

This example is hypothetical and idealized, but gives 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 they 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 together 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 behaviors that need to be defined for each instance of the role. For example, all

```
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)
```

Fig. 1. Variable-oriented pseudocode

GATHERERs receive a definition of how their values change as a function of the same variable's old value, 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, for a MOST-WANTED HOLDER a function would be defined to determine 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. Subsection 3.1 sketches out a variable-oriented programming language that uses roles of variables as language-level abstractions. Subsection 3.2 then takes a look at how a similar framework could be implemented in an existing programming language.

3.1 A Roles-Based Language

Figure 2 shows an example of variable-oriented code based on roles of variables. It is written in a speculative language called ROTFL[1]. 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'.

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

Follower prev:
    inits to: 0
    follows: curr

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

Fig. 2. The Fibonacci algorithm in the language ROTFL

[1] Role-Oriented, Titillating but Fictional Language

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 keywords related to defining or using variables with particular roles (e.g. `Follower`, `update`) are reserved words. 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 by using the reserved word `update`, which uses the `updates with` function to compute the new value for the variable, and FOLLOWERs receive new values implicitly as the value of the followed variable changes.

Traditional-style 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)
```

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

3.2 Implementing Roles in an Existing Language

Variable-oriented programming can also be done in an existing programming language, provided a suitable framework is available for this purpose. Figure 4 shows how the variable-oriented, roles-based program from Figures 1 and 2 can be written in Python.

The program in Figure 4 relies on a framework that defines roles of variables as Python classes, and role-related actions (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 Appendix A.

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

Fig. 4. A variable-oriented code fragment in Python. (Makes explicit use of lambda calculus [16].)

4 Discussion

4.1 Uses of Roles-Based Programming

According to Sajaniemi's research, the behavior of 99% of variables can be characterized with a small set of roles, at least in novice-level programs [10]. It does not immediately follow 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 very 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 findings in cognitive psychology of programming suggest that it is likely that this facilitates the extraction and construction of variable-related schemas and therefore in one way 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, and if there were roles-aware program development tools that could provide helpful feedback and error messages. There are also clearly 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 [9], which provides multiple views to 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 very self-documentative of variable-related schemas (see e.g. Figure 2). Roles of variables help in this, as role names succinctly describe patterns of variable use. It is not immediately obvious what the documentative value of variable-oriented notations is compared to non-variable-oriented notations that use code comments

to note the role of each variable. Documenting delocalized variable behavior using role names may often do enough and using a variable-oriented language may be 'overkill' for this purpose.

Even beginners are not taught variable-oriented, roles-based programming directly, they might indirectly benefit from it. In [17], Bergin 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 to hone one's skills in a particular technique and to ingrain that technique into one's thinking. Specifically, for helping instructors (not novice programmers) to 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." [17]

In a similar vein, doing 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 help 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.

4.2 Variable-oriented 'purity'

According to [9]:

> "Variable-oriented programming is a new programming paradigm which collects *all* actions concerning any single variable together. [...] [T]he plan of a variable is clearly visible and *totally* described in the variable definition." (emphases added)

A 'purely' variable-oriented program, then, would gather all references to a variable (assignments and reads) 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 (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 procedural and control-flow-oriented. It can be contrasted with the purely variable-oriented views displayed by the VOPE tool [9].

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 (Is it 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.

4.3 Compatibility with object-orientation

The original set of roles of variables was discovered by analysing procedural programs. Since then, roles of variables have been applied to object-oriented as well as functional programs [18]. 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 [9]:

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

```
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
```

Fig. 5. A ROTFL class representing simple bank accounts

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.

Another issue that needs to be considered when applying roles of variables to object-oriented programs was noted in [13]:

> "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 for instance variables are needed. For instance, 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.

5 Future Work

This paper has only 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, which, although not variable-oriented, allows the programmer to associate 'action specifications' to variables [19].

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 programs as 'etudes' 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 such as ROTFL.

6 Conclusions

In this paper, I have revisited the notion of variable-oriented programming and shown how variable-orientation can be founded on roles of variables. The paper has described ongoing work on tools for roles-based programming, and discussed the possible applications, merits and problems of the approach. I have pointed at possible research directions for the future, and it is my hope that this paper can serve as a basis for further discussions of variable-oriented, roles-based programming.

References

1. 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.: A multi-national study of reading and tracing skills in novice programmers. SIGCSE Bulletin **36** (2004) 119–150
2. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bulletin **33** (2001) 125–180
3. Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. In: Readings in artificial intelligence and software engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1986) 507–521
4. Detienne, F.: Expert programming knowledge: A schema-based approach. In Hoc, J.M., Green, T.R.G., Samurcay, R., Gilmore, D.J., eds.: Psychology of Programming. Academic Press, London (1990) 205–222
5. Rist, R.S.: Schema creation in programming. Cognitive Science **13** (1989) 389–414
6. Byckling, P., Sajaniemi, J.: 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, New York, NY, USA, ACM Press (2006) 85–96
7. Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J.: What do novices know about programming? In Badre, A., Shneiderman, B., eds.: Directions in Human-Computer Interactions. Ablex Publishing (1982) 27–54
8. Soloway, E., Lampert, R., Letovsky, S., Littman, D., Pinto, J.: Designing documentation to compensate for delocalized plans. Communications of the ACM **31** (1988) 1259–1267
9. Sajaniemi, J., Niemeläinen, A.: 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.), New York, NY, USA, Elsevier Science Inc. (1989) 66–73
10. Sajaniemi, J.: An empirical analysis of roles of variables in novice-level procedural programs. In: Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments, IEEE Computer Society (2002) 37–39
11. Sajaniemi, J.: The Roles of Variables Home Page. `http://cs.joensuu.fi/~saja/var_roles/` (2003)
12. Sajaniemi, J., Kuittinen, M.: An experiment on using roles of variables in teaching introductory programming. Computer Science Education **15** (2005) 59–82

13. Sorva, J., Karavirta, V., Korhonen, A.: Roles of variables in teaching. Journal of Information Technology Education (2007) [in press]
14. Sajaniemi, J., Navarro Prieto, R.: Roles of variables in experts' programming knowledge. In: Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG). (2005) 145–159
15. Byckling, P., Sajaniemi, J.: Roles of variables and programming skills improvement. SIGCSE Bulletin **38** (2006) 413–417
16. Lambda calculus. In: Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Lambda_calculus (retrieved June 8th, 2007)
17. Bergin, J.: Variations on a polymorphic theme: An etude for computer programming. http://www.cs.umu.se/~jubo/Meetings/ECOOP05/Submissions/Bergin-full.pdf (2005)
18. Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., Kulikova, Y.: Roles of variables in three programming paradigms. Computer Science Education **16** (2006) 261 – 279
19. Yung, E., Joy, M., Ward, A.: Eden - the Engine for DEfinitive Notations. http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/ (Retrieved April 15th, 2007)

Appendix A: 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 two roles (GATHERER and FOLLOWER). For an example of using the classes, see Figure 4.

```
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 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)
```