Active Variables in Smalltalk-80

Steven L. Messick
Kent L. Beck

CR-85-09
Computer Research Lab
Tektronix, Inc.

February 7, 1985

Active Variables in Smalltalk-80

ABSTRACT

Active values *a la* LOOPS provide a flexible means of attaching auxiliary procedures to variables which are called when the variable is is accessed or changed. The procedures, called daemons, can be used as a debugging tool, and they can be used to implement a "dials and gauges" package, which makes the value of a variable visible in various ways. This paper provides an introduction to an implementation of active variables, very similar to active values, done for Smalltalk-80. It is intended to provide the experienced Smalltalk user with a introduction to how to use the active variables package.

Active Variables in Smalltalk-80

Steven L. Messick

Kent L. Beck

CR-85-09
Computer Research Lab
Tektronix, Inc.

1. Active Variables

Several recent programming environments have provided a method for dynamically monitoring variable assignment and retrieval. LOOPS [Bobrow83] provides active values to monitor the values of variables and properties. Magpie [Delisle83] allows variables and procedures to be monitored. Standard Smalltalk, however, does not provide a good method for monitoring variables. London and Duisberg [Londone84], and Wagner [Wagner85] have employed a coding style which simulates active values in Smalltalk. The disadvantage to their technique is that any code which is to use active values must be modified to explicitly record changes to active values. We have chosen to implement active values as a feature of the language; this minimizes the impact of active values on the programmer. Our implementation is coded in Smalltalk and may be added to an existing Smalltalk image by loading the appropriate file.

We distinguish between active variables and general active objects. Active objects are objects which effect an operation which itself is not a part of the obvious flow of control which is controlling the object. As an example, an active object may invoke a daemon whenever a message is received. We have been unsuccessful in defining a general technique to activate an arbitrary object, therefore this implementation does not include support for active objects. Active variables are variables which invoke a daemon whenever they are accessed or changed. Our implementation currently supports active instance variables for objects and active temporary variables for methods. We also include a technique for advising methods, similar to the *advise* package of Interlisp [Xerox83], by allowing the insertion of arbitrary code before a method executes, or afterwards, or both.

1.1 Creating active variables

Classes with active variables must have the ability to record which instance variables are active. For this reason a slightly different metaclass is created when a class with active instance variables is defined. The way to declare a class to have an active variable (AV) is to include a line in the class definition message following "instanceVariables:" that looks like:

activeVariables: '(av1 getFn1 putFn1) (av2 getFn2 putFn2)'

The first field in the parenthesized list is the name of the AV, the second field is the message to be sent self when the value of the AV is accessed, and the third field is the message to be sent when the AV is changed.

To declare a temporary active variable in a method the temporary variables declaration of the method definition should look like:

| x y (av1 getFn1 putFn1) (av2 getFn2 putFn2) |

This makes *x* and *y* normal temporary variables and *av1* and *av2* temporary active variables. Temporary active variables may be used in any class, regardless of whether the class includes active instance variables.

1.2 Active Variables

The active values package defines three new classes, InactiveVariable, ActiveVariable, and SpecificActiveVariable. Here are their definitions and protocols:

class name	InactiveVariable
superclass	Object
instance variable names	'object'
category	'Kernal-Objects'

instance protocol

accessing

value
Return the value of the active variable.

converting

asActiveVariable
Convert an InactiveVariable to an ActiveVariable.

asInactiveVariable
Return self.

class name	ActiveVariable
superclass	InactiveVariable
instance variable names	'name getDaemon setDaemon'
category	'Kernal-Objects'

instance protocol

accessing

getDaemon
Return the selector of the method to perform when object is accessed.

getDaemon: newDaemon
Change the selector of the method to perform when object is accessed. If newDaemon is a block then convert me to a SpecificActiveVariable.

name
Return the name of the instance variable of which I am the value.

name: aString
Save the name of the instance variable of which I am the value.

setDaemon
Return the selector of the method to perform when object is assigned.

setDaemon: newDaemon
Change the selector of the method to perform when object is assigned. If newDaemon is a block then convert me to a SpecificActiveVariable.

value: val
Set the value of the active variable I represent.

converting

asActiveVariable
Return self.

asInactiveVariable
Convert me to an InactiveVariable.

private

becomeActiveVariable
I am to become an ActiveVariable. Make the necessary conversions.

becomeSpecificActiveVariable
I am to become a SpecificActiveVariable. Make the necessary conversions.

class name	SpecificActiveVariable
superclass	ActiveVariable
category	'Kernal-Objects'

instance protocol

private

becomeActiveVariable
I am to become an ActiveVariable. Make the necessary conversions

becomeSpecificActiveVariable
I am to become a SpecificActiveVariable. Make the necessary conversions.

Class InactiveVariable provides protocol to retrieve and modify its single instance variable. Instances of class ActiveVariable are used in the package as a wrapper for values of variables which are active. In addition to the protocol for accessing *object* it provides protocol to invoke the appropriate daemon when the variable is accessed. It includes instance variables *name*, *getDaemon*, and *setDaemon*. *name* is used to store the name (a String) of the variable as defined in the class using it as an active variable. *getDaemon* and *setDaemon* store a Symbol which is the selector of the method to be invoked whenever the variable is fetched or set, respectively. Class SpecificActiveVariable provides functionality similar to ActiveVariable with the difference that SpecificActiveVariable expects *getDaemon* and *setDaemon* to each hold a block of Smalltalk code. The Smalltalk block is evaluated to simulate the activation of the daemon. The only technique provided to create a SpecificActiveVariable is to send a daemon specification message to the active variable which specifies that one or both of the daemons is to be a block; conversion between ActiveVariable's and SpecificActiveVariable's occurs automatically. Note that if one daemon of a particular active variable is a block (or a method selector) then the other must also be a block (or selector).

Active variables may be made "inactive." The function of an instance of InactiveVariable is to serve as a replacement for the instance of ActiveVariable when it is deactivated. E.g. InactiveVariable's may be used to speed up some instances of a class without affecting the other instances of the class or having to recompile the methods of the class.

13 Get daemons

Get daemon methods take one argument, the active variable being accessed. Get daemon blocks require two arguments: the first is the parent of the active variable, the second is the active variable itself. Three standard get daemons are implemented in class Object, **get:** is a no-op, **transcriptShow:** writes the value of the AV to the system transcript, and **shortStack:** writes both the value of the AV and a short context stack on the transcript. If no daemon is specified for an active variable then **get:** will be used by default.

The value that a get daemon returns is the value that is used as the value of the access to the active variable, so in general get daemon methods look like:

```
genericGetDaemon: activeValue
        .
        .
        .
'activeValue value
```

Blocks which are functioning as get daemons resemble:

```
[ :parent :activeValue |
        .
        .
        .
activeValue value]
```

Since the responsibility of determining the value of an active variable rests entirely within the get daemon, the value of an active variable may be an arbitrary function.

The following class demonstrates the two kinds of active variables, instance and temporary. First we declare *randomValue* to be active, with the get daemon *randomGetDaemon*. *randomGetDaemon* assumes that the value of the active variable that is passed to it is an instance of **Random**, and returns the result of sending the value the message *next*. The method *test1* initializes the instance variable *randomValue* and then uses it three times in a row. When you run this example you will notice that the values returned are all different random numbers. The method *test2* is the same, except it uses a temporary active variable.

```
Object subclass: #Test
    instanceVariableNames: 'randomValue '
    activeVariables: '(randomValue randomGetDaemon: nil) '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Gauges'

Test methodsFor: 'daemons'
    randomGetDaemon: av
        "Make the value of av into a new random
        number, then return it."

        ^av value next
```

```
Test methodsFor: 'examples'
    test1
        randomValue ← Random new.
        ↑Array with: randomValue with: randomValue with: randomValue
    test2
        | (x randomGetDaemon: nil) |
        x ← Random new.
        ↑Array with: x with: x with: x
```

1.4 Set daemons

Set daemon methods take two arguments: the first is the active variable which is to be modified, the second is the new value of it. Set daemon blocks require three arguments: the first is the parent of the active variable, the second is the active variable itself, and the third is the new value of the active variable. The set daemons which correspond to the predefined get daemons are **set:to:**, **transcriptShow:setTo:**, and **shortStack:setTo:**. The default set daemon is **set:to:**. Consistent with get daemons, set daemons are responsible for actually changing the value of the active variable. The template for a set daemon method is:

```
genericSetDaemon: activeVariable setTo: newValue
        .
        .
        .
activeVariable value: newValue
```

The template for a set daemon block is:

```
[ :parent :activeVariable :newValue |
        .
        .
        .
activeVariable value: newValue]
```

Below is an example of the use of setDaemons in a context similar to the previous example. In this case when randomValue is initialized the value it is to be set to is checked to gaurentee that it is an instance of **Random**, if not it is coerced.

```
Object subclass: #Test
    instanceVariableNames: 'randomValue '
    activeVariables: '(randomValue randomGetDaemon: random:setTo:) '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Gauges'

Test methodsFor: 'daemons'
    randomGetDaemon: av
        "Make the value of av into a new random
        number, then return it."

        ↑av value next
```

```
random: av setTo: newValue
        "Make the value of av be newValue unless newValue
        is not an instance of Random."

        newValue isMemberOf: Random
                ifFalse: [tav value: Random new].
        tav value: newValue
Test methodsFor: 'examples'
    test1
        randomValue ← 0.        "force the daemon to do coersion"
        tArray with: randomValue with: randomValue with: randomValue
    test2
        | (x randomGetDaemon: nil) |
        x ← Random new.
        tArray with: x with: x with: x
```

1.5 Additional Protocol for Active Variables

It is possible to manipulate active variables in ways independent of their daemons. Before describing these functions we must first provide a few implementation details.

The mechanism by which variables are activated is provided by the Smalltalk compiler. Each assignment to an active variables is converted to a message sent to the active variable with a parameter which is the new value the active variable is to be set to. Similarly, active variable fetches are converted to a (different) message sent to the active variable. Each of these messages includes a parameter which is the object who "owns" the active variable. (I.e. the parent of the active variable; this object is always referred to by "self" during compilation of references to active variables.) When the active variable receives the message it, in turn, sends the appropriate get or set daemon message to its "owner" if it is a method; blocks daemons acquire access to the parent object via an explicit argument to the block.

Obviously, all variables which are active must be instantiated to an instance of ActiveVariable if this mechanism is to work. It is for this reason that a class which uses active instance variables must be declared with active variables. Metaclass definitions for classes with active variables include an additional instance variable *activeVariables*. This instance variable stores all initialization information for active instance variables. Initialization of instance variables is accomplished at instance creation time: **basicNew** has been defined to initialize active variables when a new instance is created. Any modifications to **new** should use **basicNew** to actually create the new instance. Temporary active variables are initialized when the method defining them is entered. The compiler inserts the necessary initialization code at the beginning of each method which uses temporary active variables.

The classes Class, ClassDescription, Behavior, and Object include some additional protocol to support active variables.

Behavior instance protocol

active variables

allActiveVariables
Return a structure containing a variable name, its index, and daemon names for each active variable in the receiver and all its superclasses.

Class instance protocol

subclass creation

subclass:instanceVariableNames:activeVariables:classVariableNames:...
This is the standard class creation message, extended to support active variables.

variableByteSubclass:instanceVariableNames:activeVariables:...

variableSubclass:instanceVariableNames:activeVariables:...

variableWordSubclass:instanceVariableNames:activeVariables:...

Class class protocol

subclass creation

named:superclasses:instanceVariableNames:activeVariables:...
This is the multiple-inheritance class creation message, extended to support active variables.

ClassDescription instance protocol

active variables

activeVariablesString
Return a string which correctly defines each active variable of the receiver. This is for use in printing a class definition.

activeVarNames
A misnomer. Return a structure with the same form as that of BehaviorlallActiveVariables but only for the active variables defined in the receiver.

allSelectorsWhichAccess: anActiveVar
Return the selector and class of all methods which use the instance variable anActiveVar in the receiver and all its subclasses. This is primarily for use in "activating" and "inactivating" instance variables.

hasActiveVariables
Return true if the receiver has active variables. This method is redefined in each class defined which uses active variables.

makeActiveVariable: instVar
Make the instance variable instVar of the receiver an active

variable.

makeInactiveVariable: anActiveVar
Make the active instance variable anActiveVar of the receiver a normal instance variable.

set: aVar getDaemon: daemonName
Set the get daemon of aVar to be daemonName. If aVar is not active then activate it. This method does not affect existing instances of the receiver.

set: aVar getDaemon: getDaemonName setDaemon: setDaemonName
Set the daemons of aVar to getDaemonName and setDaemonName. Activate aVar if it is not active. This method does not affect existing instances of the receiver.

set: aVar setDaemon: daemonName
Set the set daemon of aVar to be daemonName. If aVar is not active then activate it. This method does not affect existing instances of the receiver.

Object instance protocol

active variables

initializeActiveVariablesFrom: anArray
Initialize the active instance variables of the receiver. This method, by default, is sent from basicNew when an instance is created. It should not be used elsewhere.

set: aVar getDaemon: daemonName
Set the get daemon of aVar to be daemonName in the receiver only. aVar must be active.

set: aVar getDaemon: getDaemonName setDaemon: setDaemonName
Set the daemons of aVar in the receiver. aVar must be active.

set: aVar setDaemon: daemonName
Set the set daemon of aVar to be daemonName in the receiver only. aVar must be active.

The methods **set:getDaemon:**, **set:getDaemon:setDaemon:**, and **set:setDaemon:** are used to change the daemon(s) for a particular instance of an object. If the daemon is a block then the ActiveVariable will be converted to a SpecificActiveVariable.

Object instance protocol

active var daemons

get: anActiveVar
Return the value of anActiveVar. This is the system default get daemon.

set: anActiveVar to: aValue
Set the value of anActiveVar to be aValue. This is the system default set daemon.

shortStack: anActiveVar
Display the top five elements of the context stack on the Transcript. Return the value of anActiveVar.

shortStack: anActiveVar setTo: aValue
Display the top five elements of the context stack on the Transcript. Set the value of anActiveVar to be aValue.

transcriptShow: anActiveVar
Display the value of anActiveVar on the Transcript. Return its value.

transcriptShow: anActiveVar setTo: aValue
Display a message indicating the the value of anActiveVar changed to aValue. Set the value of anActiveVar to be aValue.

2. Dials and Gauges

LOOPS includes a set of dials, meters, and scales which may be used to graphicly monitor the value of active variables. This has inspired us to write our own dials and gauges package to use with Smalltalk active variables. The only gauge implemented so far is called LCD. Using the class Test from above, to created an LCD you would say:

```
t ← Test new.
l ← LCD onVar: 'randomValue' of: t.
l open
```

Then, when you say:

```
t randomValue: 10.
t randomValue: 50
```

The LCD first displays 10, then 50.

We plan to implement a full range of dials and gauges including, but not limited to: thermometers, sliding scales, speedometers, and anything else that may be useful.

3. Advised methods

The advised method portion of this package allows the user to insert code before and after a given method, without having to recompile the method. It is useful for setting conditional breakpoints, for argument and result type checking or coersion, for tracing, and for changing the interface to a method, all without having to change the method.

We borrowed the idea for advise from Interlisp, and we will adopt their terminology here. The visible results of loading the advised method package is the addition of two options to the protocol menu of all browsers.

advise
: This option opens an advise browser containing a template of the advising code or the previously compiled advise code. Choosing *accept* within this code will advise (or readvise) the function originally chosen to be advised.

unadvise
: This option removes advise from a method.

Some caveats: You are likely to have strange things happen if you halt within a method that is advised. Funny stuff will also occur if you manually remove the advising methods (found in protocol "advising methods") -- these are created with the *advise* menu option; use the *unadvise* menu option to effect cleanup.

4. Summary

We have described two enhancements to the Smalltalk programming environment: active variables, which provide a "value-oriented" approach [Stefik83], and active methods, which allow sections of code to be "advised" similar to advised functions in Interlisp.

5. REFERENCES

[Bobrow83] Bobrow, D. G., and M. Stefik, "The LOOPS Manual," Xerox Palo Alto Research Center, December 1983.

[Delisle83] Delisle, N. M., D. E. Menicosy, and M. D. Schwartz, "Magpie -- An Interactive Programming Environment for Pascal," TR CR-83-4, Tektronix, Inc., Beaverton, Oregon, 1983.

[London84] London, R. L., and R. A. Duisberg, "Animating Programs Using Smalltalk," Computer Research Lab., Tektronix, Inc, Beaverton, Oregon, 1984.

[Stefik83] Stefik, M., D. G. Bobrow, S. Mittal, and L. Conway, "Knowledge Programming in LOOPS: Report on an Experimental Course," *The AI Magazine*, IV 3, 1983, pp3-13.

[Xerox83] *Interlisp Reference Manual*, Xerox Palo Alto Research Center, October, 1983.

[Wagner85] Wagner, R., Master's Thesis (in preparation), MIT, 1985.