Bob Hinkle Ralph E. Johnson

The active life is the life for me!

ALMOST EVERYTHING IN Smalltalk is an object, including language elements like classes, methods, processes, and contexts. Variables are a major exception to this rule of thumb. While global and class variables are objects in most implementations of Smalltalk, instance variables and temporary variables are not. That's too bad, because instance variables have many uses as objects. (Making temporary variables first-class seems less useful.)

For example, user interface widgets often wish to depend on a particular attribute of an object. VisualWorks has you represent these attributes with ValueHolders, which are objects that hold a single value. If you store an attribute in a ValueHolder, widgets can depend on the ValueHolder and be notified when that attribute changes. However, to change a design that doesn't use ValueHolders to one that does, you have to rewrite your program, changing accesses to attributes stored in ValueHolders. There are times when you want to keep the program as it is, but just change the way you store the attribute. In other words, you'd like to depend directly on a variable without explicitly having to store a ValueHolder in it. This feature is called active variables* and is very useful when you are debugging and fine-tuning a program.

In this column we use VisualWorks 2.0 to implement active variables in three steps. First, we define the class ActiveVariable and show how to convert an object's slots to contain instances of it. Next, in the largest step, we introduce a new class for an object containing one or more ActiveVariables. Finally, we use a new kind of MethodProducer to recompile methods in the new class.

Bob Hinkle is an independent Smalltalk consultant and writer. His current focus is the improvement of existing tools and the creation of new tools to revitalize the Smalltalk environment. He can be reached at hinkle@primenet.com.

Ralph Johnson learned Smalltalk from the *Blue Book* in 1984. He wrote his first Smalltalk program in the fall of 1985 when he taught his first course on object-oriented programming and design. He has been a fan of Smalltalk ever since. He is the only author of "*Design Patterns: Elements of Reusable Object-Oriented Software*" to regularly program in Smalltalk, and continues to teach courses on object-oriented programming and design at the University of Illinois.

ACTIVE IS AS ACTIVE DOES

To our knowledge, the first language to include active variables was LOOPS. Active variables descended to CLOS in the form of access daemons on slots. We're not the first to implement active variables in Smalltalk (see Messick[1] for example), and our specification is similar to what others have done. Our implementation of ActiveVariables is new, since we use the GenericCompiler and MethodProducer described in our previous columns. ActiveVariables satisfy the following high-level specification:

Class: ActiveVariable
Superclass: Object
Important instance variables:
name <String>
value <Object>
readDependents <Set of 2-argument BlockClosures>
writeDependents <Set of 3-argument BlockClosures>
Important instance methods:
value

Return the ActiveVariable's value, and also notify readDependents.

value: anObject

Set the ActiveVariable's value, and also notify writeDependents.

The read dependents are two-argument blocks, whose parameters are the ActiveVariable and its current value. These blocks are evaluated whenever the message #value is sent to the ActiveVariable, as follows:

Method for ActiveVariable

```
value
    readDependents do: [ :each | each value: self value:
        value].
    ^value
```

Similarly, the write dependents are evaluated whenever

* Source code for the active variables package is available by anonymous ftp from st.cs.uiuc.edu. Look for the file ActiveVariables20.st in pub/st_vw.

#value: is sent to the ActiveVariable. They are three-argument blocks, whose parameters are the ActiveVariable, its current value, and its old, over-written value:

Method for ActiveVariable

```
value: anObject
    "This method needs to return the new (i.e., being set)
     value, so the behavior is consistent when an
     ActiveVariable set replaces a := expression."

   | oldValue |
   oldValue := value.
   value := anObject.
   writeDependents do: [ :each | each value: self value:
   value value: oldValue].
   ^value
```

ActiveVariables are added to an object using #instVarAt:put:, the primitive method that gives direct access to object state. While this method is a gross violation of encapsulation, it's required for Smalltalk programming tools written in Smalltalk, such as the Inspector. We use the phrase "activating an object" to describe the process of adding ActiveVariables to the object's slots, and similarly define an "activated object" to be an object that has ActiveVariables implicitly stored in some of its slots. Once an object has been activated, its class must be changed, and ideally the slot conversion and class change should happen atomically, to prevent access errors. The object's new class has methods re-compiled to send #value and #value: to access any activated slots. We show how to create such a class in the next two sections.

ACTIVATED OBJECTS NEED ACTIVE CLASSES

Since an activated object needs specially compiled methods, it also needs a special class to store those methods. Furthermore, this class must be distinct for different objects, so that you can activate one of a class's instances without activating all of them. This kind of class, which is distinct for individual objects, is called a lightweight class, which we showed how to implement in *Debugging Objects*.[2] Activated objects are instances of ActiveClass, which is a new subclass of LightweightClass that adds specialized support for ActiveVariables. In addition, each ActiveClass needs some information for each of its variables that is activated. A new object called ActiveVariable-Specification maintains this information for the ActiveClass. These two new classes are specified by

Class: ActiveClass
Superclass: LightweightClass
Important instance variables:
baseClass <Class>
activeVariables<Collection of ActiveVariableSpecifications>
Important instance methods:
activateObject: <Object>
Create and install ActiveVariables in slots of the Object corresponding to the receiver's ActiveVariableSpecifications.
activeVariables
activeVariableIndexes
Return respectively the collection of names and indexes of this ActiveClass' activated variables.
allActiveVariables
allActiveVariableIndexes
Return respectively the collection of names and indexes of all activated variables in this ActiveClass and all of its transitive superclasses.
convertInstancesTo: <ActiveClass> addedIndexes: <Collection of SmallIntegers> from: <Behavior>
Convert any instances of the receiver to have the input ActiveClass as their class. This involves changing their class, adding ActiveVariables to any slots numbered in the input collection, and recompiling methods that reference those slots.
noteNewActiveVariables: <Collection of ActiveVariableSpecifications>
Stores any new ActiveVariableSpecifications in the activeVariables instance variable and converts the receiver's instances to support these specifications.
Important class methods:
destroyAllActiveClasses
Eliminates all active classes and their instances
Class: ActiveVariableSpecification
Superclass: Object
Important instance variables:
name <String>

index <SmallInteger>	
globalReadDependents	<OrderedCollection of 2-argument BlockClosures>
globalWriteDependents	<OrderedCollection of 3-argument BlockClosures>
localReadDependents	<IdentityDictionary mapping objects to 2-argument BlockClosures>
localWriteDependents	<IdentityDictionary mapping objects to 3-argument BlockClosures>

Important instance methods
addDependentsFrom: <ActiveVariableSpecification>
Merges the ActiveVariableSpecification's dependents lists into the receiver's.
setDependentsOf: <Object>

Adds dependent blocks to the ActiveVariable in the Object's slot indicated by the receiver's index instance variable. All blocks from the global lists are added, as well as any associated with the Object in the local dictionaries.

setReadDependents: <Collection of 2-argument BlockClosures>
setWriteDependents: <Collection of 3-argument BlockClosures>

Respectively set the read- and write-dependent collections for this specification.

ActiveClasses create a new distinction between two kinds of

LightweightClass. Originally, a LightweightClass acted as an extension of an object's original class. The LightweightClass stood between the object and its original class, allowing the object to inherit methods from the original class as long as they weren't over-ridden in the LightweightClass. Most ActiveClasses will not be extensions of the original class, but instead they will completely replace the original class in the object's look-up chain.

Figure 1 illustrates these two different arrangements. In this example, aSet1 and aSet2 were both instances of Set, whose superclass is Collection. An extension LightweightClass named ExtensionToSet was created for aSet1 and inserted between aSet1 and Set. A replacement LightweightClass named ReplacementForSet was created for aSet2 and inserted between it and Collection, effectively removing Set from aSet2's lookup chain. aSet1 responds to #size using the method defined in Set, its original class, since that method is not over-ridden in ExtensionToSet. However, it responds to #isEmpty using the method defined in ExtensionToSet (which could still refer to Set's method #isEmpty by a super send). On the other hand, aSet2 responds to #size using the method from Collection. It responds to #isEmpty using the method defined in ReplacementForSet, which cannot access Set's method #isEmpty since Set is nowhere on Replacment ForSet's superclass chain.

There are two reasons why ActiveClasses are best implemented as replacement LightweightClasses. First, significant space can be saved when objects are activated from several classes in one class hierarchy. If ActiveClasses were extensions, and a DependentPart object and a CompositePart object were activated, the DependentPart's ActiveClass and the CompositePart's ActiveClass would both have recompiled versions of methods from their common superclasses. With ActiveClasses implemented as replacements, though, the two ActiveClasses would both inherit from an ActiveClass for VisualPart, eliminating that method duplication. The second justification is that extension classes must change the way they compile methods that use super. Using the same example, if a VisualPart method referencing super were naively recompiled in the DependentPart's ActiveClass, super would refer to VisualPart and not Object. Using replacements, the ActiveClass hierarchy parallels the original class hierarchy, guaranteeing that the meaning of super in recompiled methods will remain correct.

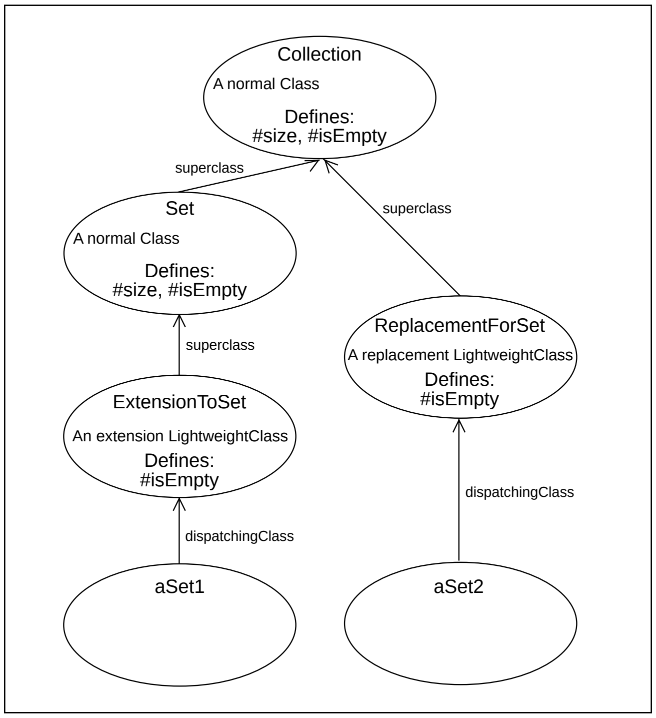


Figure 1. The difference between extension and replacement LightweightClasses.

Using replacements instead of extensions doesn't remove all technical difficulties, of course. For example, suppose three objects with the same class are activated, only each activates a different subset of the class's variables. How many ActiveClasses should be created? On one extreme, we could create three ActiveClasses, one for each object. Should another object of the same class be activated in exactly the same slots, it could share the ActiveClass of one of these three, but otherwise a new ActiveClass would be required. This answer guarantees that each ActiveClass minimally matches its instances' needs for activation, but also has the potential to waste space. On the other extreme is the solution we've adopted, which uses only one ActiveClass for every class. In the above scenario, this ActiveClass will activate all slots needed by any one of its activated instances. As a result, some objects will have variables activated unnecessarily. This will not cause any incorrect behavior, though it will unnecessarily (but insignificantly, in our opinion) slow the objects' access to their variables. As another result, a new instance variable named activeVersion is added to Behavior, and thus to every existing and added class in the system. This variable will either hold the Behavior's unique active version or nil if it has none. In addition, we've added a number of new methods to Behavior. These new additions are summarized by

Class: Behavior
Added instance variables:
activeVersion <nil | ActiveClass>
Added instance methods
activateObject: <Object>
variableIndexes: <Collection of SmallIntegers>
readDependents: <Collection of 2-argument
BlockClosures>
writeDependents: <Collection of 3-argument
BlockClosures

Installs ActiveVariables in all slots of the Object indicated in the #variableIndexes: parameter. Adds the given read- and write-dependents to these ActiveVariables. Changes the Object's class to an appropriate ActiveClass.

activeSuperclass

If the Behavior's activeVersion isn't nil, returns the active Version's superclass. Otherwise returns the Behavior's superclass.

```
buildActiveClassForVariables: <Collection of
                               ActiveVariableSpecifications>
buildActiveClassForVariables: <Collection of
                               ActiveVariableSpecifications>
                               from: <Behavior>
```

Do whatever work is necessary, which can sometimes be quite a lot, to build an ActiveClass that activates all slots indicated by the input collection.

```
buildActiveSuperclassForVariables:
    <Collection of ActiveVariableSpecifications>
```

Obtains an ActiveClass for the Behavior's superclass that activates any variables in the input collection that are defined in that superclass or higher.

You activate an object by sending it #activateVariables: readDependents:writeDependents:. The first parameter is a collection of variable names to be activated, and the last two respectively are collections of read- and write-dependent blocks to be registered for all the slots being activated. The implementation in Object forwards the message to Behavior, which implements it as:

Method for Behavior

```
activateObject: object variableIndexes: indexCollection
readDependents: readBlocks writeDependents:
writeBlocks
    | specs activeClass names |
    names := self allInstVarNames.
    specs := indexCollection collect: [ :each |
        self activeVariableSpecClass new
            name: (names at: each) index: each;
            setReadDependents: readBlocks for: object;
            setWriteDependents: writeBlocks for: object].
    activeClass := self buildActiveClassForVariables: specs.
    activeClass activateObject: object
```

This method has three major steps, corresponding to its last three statements. The #collect: loop creates an Active VariableSpecification for each slot. A dependent registered with an ActiveVariableSpecification can be associated with one particular activated object, as is done above, or it can be registered globally. With the latter option, you can monitor all accesses to a particular variable across a group of activated objects. The second step sends #buildActiveClass ForVariables: to create an ActiveClass that re-compiles all methods referencing variables in index Collection. The final statement sends #activateObject: to the active class, which is defined as:

Method for ActiveClass

```
activateObject: object
    | oldActiveVariableIndexes newActiveVariableIndexes
      names |
    oldActiveVariableIndexes := object dispatchingClass
                                 allActiveVariableIndexes.
    (newActiveVariableIndexes := self
     allActiveVariableIndexes)
        removeAll: oldActiveVariableIndexes.
    names := baseClass allInstVarNames.
    newActiveVariableIndexes do: [ :each |
        object instVarAt: each put:
            (self activeVariableClass
                name: (names at: each)
                initialValue: (object instVarAt: each))].
    object changeClassToThatOf: self basicNew.
    self allActiveVariables do: [ :each |
        each setDependentsOf: object]
```

The bulk of this message (up to the #changeClassToThatOf: send) creates and installs ActiveVariables in the newly activated slots of the input object. It's important to install ActiveVariables only in newly activated slots, since otherwise the code could produce limitless chains of nested ActiveVariables. The initial value for each new ActiveVariable is the old (non-active) value for the corresponding slot. After the ActiveVariables are installed, the receiver becomes the activated object's new class (and, as we mentioned before, this should ideally happen atomically with the slot conversion). Finally, the ActiveVariableSpecifications help initialize the newly activated object by copying their read- and write-dependents for that object, as well as all global dependents.

Behaviors respond to the #buildActiveClassForVariables: message used above by sending themselves #buildActive ClassForVariables:from: with the second parameter equal to self. This method constructs an ActiveClass for a given set of ActiveVariableSpecifications, which may require creating ActiveClasses for superclasses or converting existing ActiveClasses to support newly activated slots. Converting an existing ActiveClass may require work on its subclasses, since newly activated slots require the installation of ActiveVariables and the re-compilation of all referencing methods in the class's instances and also in the instances of any classes that inherit from it. Thus, the method that does the conversion (which has the longish name #convertActiveClassWithSuperclass:addNewActiveVariables:from:) calls back to #buildActiveClassForVariables:from: again. The sendingClass is passed in to ensure that every class gets converted only once.

Method for Behavior

```
buildActiveClassForVariables: variableSpecs from:
sendingClass
    | newActiveSuperclass |
    newActiveSuperclass := self buildActiveSuperclass
                           ForVariables: variableSpecs.
    self activeClass notNil
        ifTrue: [self convertActiveClassWithSuperclass:
                 newActiveSuperclass
                  addNewActiveVariables: variableSpecs
                  from: sendingClass]
        ifFalse: [self buildActiveClassWithSuperclass:
                  newActiveSuperclass
                   withVariables: variableSpecs].
    ^self activeClass
```

#buildActiveSuperclassForVariables: generates an ActiveClass for the superclass if it defines any of the activated slots. It does so with another send of #buildActiveClassForVariables: from:, passing in the receiver as the sendingClass. The superclass needs to know who originated the recursive message to eliminate redundant operations during its activation.

Method for Behavior

```
buildActiveSuperclassForVariables: variableSpecs
    | superclassActivatedSlots |
    superclassActivatedSlots :=
        variableSpecs select: [ :each | each index between:
1 and: self superclass instSize].
    ^superclassActivatedSlots isEmpty
        ifTrue: [
            self activeSuperclass]
        ifFalse: [
            self superclass
                buildActiveClassForVariables:
superclassActivatedSlots
                from: self]
```

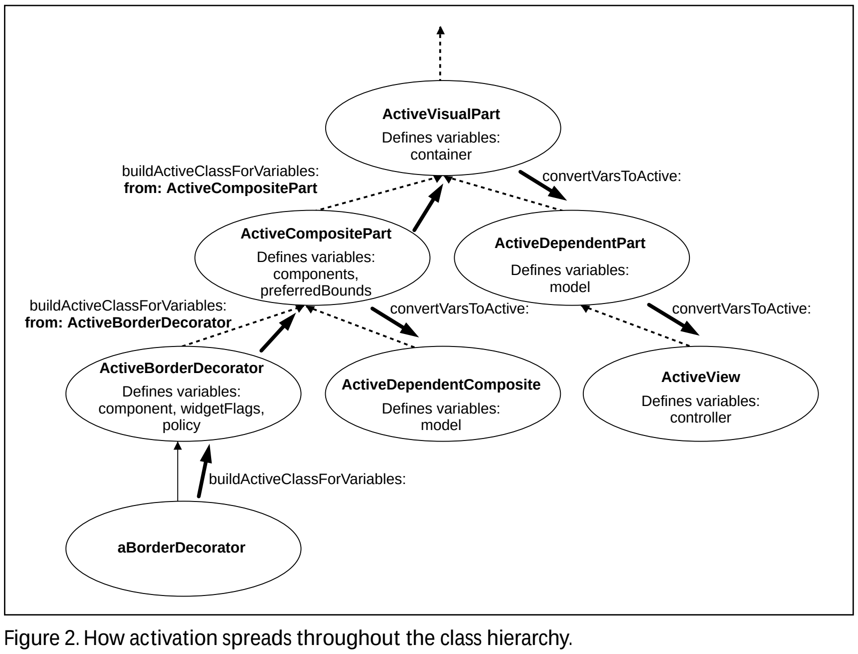
The method #convertActiveClassWithSuperclass:addNewActiveVariables:from: is called from #buildActiveClassForVariables: from: when the receiver already has an activated version. It builds a new ActiveClass, figures out exactly which variables are newly activated, and then converts instances of the old ActiveClass to the new format using #convertInstancesTo:addedIndexes:from:.

Method for Behavior

```
convertActiveClassWithSuperclass: newSuperclass
addNewActiveVariables: variableSpecs from: sendingClass
    | myActivatedSlots activeClass newActiveClass
addedIndexes |
    myActivatedSlots :=
        variableSpecs select: [ :each | each index between:
self superclass instSize + 1 and: self instSize].
    activeClass := self activeClass.
    myActivatedSlots := myActivatedSlots, (newSuperclass
variableSpecsNotIncludedIn: activeClass superclass).
    newActiveClass := activeClass copy.
    newActiveClass
        assignSuperclass: newSuperclass;
        noteNewActiveVariables: myActivatedSlots.
    addedIndexes := myActivatedSlots asOrderedCollection
collect: [ :each | each index].
    addedIndexes removeAll: activeClass
allActiveVariableIndexes ifAbsent: [].
    activeClass
        convertInstancesTo: newActiveClass
        addedIndexes: addedIndexes
        from: sendingClass.
    activeVersion := newActiveClass
```

The variable myActivatedSlots contains the activated variables that are defined in the receiver, plus any newly activated slots in newSuperclass. We need to allow for active superclasses that activate more variables than we request since every Class has just one ActiveClass version. For example, suppose we activate the variable "icon" in a ScheduledWindow and later activate the variables "application" and "label" in an ApplicationWindow. ApplicationWindow defines "application", but its superclass ScheduledWindow defines "label", so the activation process has to obtain an ActiveClass for ScheduledWindow using #buildActive SuperclassForVariables:. This ActiveClass will activate "label", as desired, but it will also activate "icon" because of the previous request. This technical point is an easy one to overlook in an implementation. (Or so we'd like to think, since we missed it in our earliest efforts!)

Figure 2 illustrates how activation spreads around the class hierarchy and why it's important to identify the source of the #buildActiveClassForVariables:from: message. In the diagram, the dashed arrows represent the superclass relationship, and the thick arrows represent message sends. If "widgetFlags", "components", and "container" are activated for aBorderDecorator, activation must spread up from ActiveBorderDecorator to ActiveCompositePart, where "components" is defined, and to ActiveVisualPart, where "container" is defined. When ActiveCompositePart recompiles methods to activate "container" and "components", it must also update all of its instances and all of its transitive subclasses' instances. However, these activation echoes must spread away from the ActiveBorderDecorator ActiveCompositePart-ActiveVisualPart branch, since their methods and instances will be converted as a


ActiveVisualPart
Defines variables: container
buildActiveClassForVariables: from: ActiveCompositePart
convertVarsToActive:
ActiveCompositePart
Defines variables: components, preferredBounds
ActiveDependentPart
Defines variables: model
buildActiveClassForVariables: from: ActiveBorderDecorator
convertVarsToActive:
convertVarsToActive:
ActiveBorderDecorator
Defines variables: component, widgetFlags, policy
ActiveDependentComposite
Defines variables: model
ActiveView
Defines variables: controller
buildActiveClassForVariables:
aBorderDecorator
Figure 2. How activation spreads throughout the class hierarchy.

result of #buildActiveClassForVariables:from: sends active for each of them. Thus, when ActiveCompositePart activates "container" and "components", it will send #convertVarsToActive: to ActiveDependentComposite but not to ActiveBorderDecorator. Similarly, when ActiveVisualComponent activates "container", it will send #convertVarsToActive: to ActiveDependentPart (and thus indirectly to ActiveView) but not to ActiveCompositePart.

The method #convertInstancesTo:addedIndexes:from: is used when an existing ActiveClass is changed by the addition of new ActiveVariables. It converts all the instances of the old ActiveClass to the new one. The argument indexCollection indicates the newly activated variables.

Method for ActiveClass

```
convertInstancesTo: newActiveClass addedIndexes:
indexCollection from: sendingClass
    | templateObject instances |
    (instances := self allInstances) isEmpty
        ifFalse: [
            templateObject := newActiveClass basicNew.
            self convertIndexesToActive: indexCollection in:
                                            instances.
            instances do: [ :each |
                each changeClassToThatOf: templateObject].
            newActiveClass updateInstanceDependentsIn:
            instances].
    self subclasses do: [ :each |
        each baseClass == sendingClass
            ifFalse: [
                each assignSuperclass: newActiveClass.
                each withAllSubclasses do: [ :sub |
                    sub updateInstancesForActivatedIndexes:
                    indexCollection]]]
```

The message #convertIndexesToActive:in: cycles through the passed-in collection and creates an ActiveVariable for each newly activated slot. Each instance then has its class changed to the new ActiveClass. The #updateInstanceDependentsIn: message iterates over the old ActiveClass's instances and adds the dependents from its ActiveVariable Specifications. Finally, the old active class moves its subclasses (except for sendingClass) to the new ActiveClass. It changes their superclass, and then has them and all of their subclasses acti-vate the appropriate slots and recompile methods that reference any newly activated slots. This step, finally, is where the sendingClass parameter is used. The sendingClass shouldn't be converted, but only other subclasses of the old ActiveClass. The sendingClass subclass itself is processed in a #activeClassForVariableSpecs:from: context lower down in the execution stack. When that context resumes, that sendingClass will be converted appropriately, so it's vital (to avoid duplicate ActiveVariables, for example) that it not be converted here. Thus, as figure 2 showed, when class activation flows up the class hierarchy, any activation echoes flowing back down the hierarchy must flow away from classes passed up as sendingClass parameters.

The method #buildActiveClassWithSuperclass:withVariables: is called from #buildActiveClassForVariables:from: when the receiving Behavior doesn't already have an active version.

Method for Behavior

```
buildActiveClassWithSuperclass: newSuperclass
withVariables: variableSpecs
    | myActivatedSlots |
    myActivatedSlots := variableSpecs select: [ :each |
        each index between: self superclass instSize + 1
and: self instSize].
    activeVersion := (ActiveClass newWithBase: self)
                        copyAllMethods.
    myActivatedSlots := myActivatedSlots, (newSuperclass
                            variableSpecsNotIncludedIn:
                            activeVersion).
    activeVersion
        assignSuperclass: newSuperclass;
        noteNewActiveVariables: myActivatedSlots
```

In ActiveClass>>copyAllMethods, the new ActiveClass copies all methods defined in the receiving Class, which is its base class. The new ActiveClass inherits from the superclass determined earlier. In #noteNewActiveVariables:, the newActiveClass stores ActiveVariableSpecifications in its activeVariables instance variable and recompiles methods accessing activated slots to use #value and #value:.

THE ACTIVE COMPILER

The process of converting active classes relies on the #recompileMethodsReferencing: message, which is called whenever an ActiveClass' set of ActiveVariableSpecifications changes.

Method for ActiveClass

```
recompileMethodsReferencingAny: indexCollection
    | m |
    self selectors do: [ :each |
        m := self compiledMethodAt: each.
        (m usesAny: indexCollection)
            ifTrue: [self recompile: each]]
```

This method recompiles any method of the receiver that uses, by reading or by writing, any named slot whose index is in the parameter indexCollection. This recompilation ensures that all activated slots are accessed only by #value and #value: sends. These #value and #value: accesses are inserted by a special object called ActiveProducer.

ActiveProducer subclasses from the MethodProducers we discussed in our previous two columns. It introduces a new name scope, ActiveNameScope, which in turn creates an instance of ActiveLocalScope for any ActiveClass. During compilation, local name scopes create instances of subclasses of class Variable for each name in a parse tree. The choice of subclass affects the code generated to access the variable. For example, InstanceVariable emits primitive bytecodes for direct instance access. StaticVariable, which is used for global names, emits more specialized code. When the compiler encounters a write to a variable, it sends #emitStorePop:value:from: to the corresponding Variable object. StaticVariable implements this method as:

```
emitStorePop: codeStream value: value from: assignment
    "Emit code to assign a value to the variable."

    self checkStore: codeStream from: assignment.
    codeStream pushConstant: binding.
    value emitValue: codeStream forAssignment:
    assignment.
    codeStream noteSourceNode: assignment.
    codeStream sendNoCheck: #value: numArgs: 1.
    codeStream pop
```

The underlined statements are the ones that actually generate code. The first generates bytecodes to push the StaticVariable's binding, which will be an Association, onto the stack. The next causes the assigned value to generate its own bytecodes. The third statement generates a #value: message send, which will store the assigned value into the Association's value instance variable. ActiveLocalScope instantiates a new class, ActiveInstanceVariable, to represent active named slots. ActiveInstanceVariable overrides the load- and store-emitting messages in a way similar to StaticVariable. For example:

```
emitStorePop: codeStream value: value from: assignment
    "Emit code to assign a value to the variable."

    self checkStore: codeStream from: assignment.
    codeStream putLoadInst: index scope: scope.
    value emitValue: codeStream forAssignment:
assignment.
    codeStream noteSourceNode: assignment.
    codeStream sendNoCheck: #value: numArgs: 1.
    codeStream pop
```

This is the same as StaticVariable's implementation except for the underlined statement. In contrast to the StaticVariable above, the ActiveVariable is accessed using the normal load for an instance variable. Since this method generates a #value: send, the #value: message will be sent to the ActiveVariable at run-time, allowing it to store the new value in its value instance variable and to alert its writeDependents.

Because StaticVariables are compiled specially, they also require special handling in the Decompiler. ActiveInstanceVariables don't require this as implemented, since they are only supported by ActiveClasses, which never decompile their methods. If you are interested in supporting ActiveVariables on Classes, you will have to add analogous special handling for ActiveVariables to the Decompiler.

CONCLUSION

The implementation we've described can be used to monitor instance variable accesses on a per-object basis. That's useful in its own right, and in addition we'll use active variables in our next column to implement watchpoints. Our watchpoints will be arbitrary expressions you can enter into a Debugger, which will then alert you if and when the value of the watchpoint expression ever changes.

There are a few limitations of our implementation that bear further investigation. Most notably, once an active version of a class is created, there is no connection between methods in the Class and its ActiveClass. Thus, you might change methods in or add methods to the class, but these changes would not be reflected in the ActiveClass. As a result of this limitation, you will have to purge ActiveClasses periodically after you've made programming changes. Some day we hope there will be a detailed dependency mechanism that would alert interested parties whenever a class' definition changes. (In fact, such a mechanism is one of the many potential projects for a future column.) In that case, we could use the dependencies to keep ActiveClasses in synch with their base class. Another extension that you may find interesting to attempt is to support the activation of indexed instance variables as well as named instance variables. In some ways this is an easier problem, since indexed variables can only be accessed via messages, but there are still some interesting issues that must be resolved to integrate indexed variables into our scheme.

We have three main goals when writing these columns. First, we try to present projects that can help improve the quality and productivity of Smalltalk programmers, and we hope that you find these ActiveVariables useful for exploring and debugging your code. Second, we try to do some interesting and perhaps unusual Smalltalk programming, to give you an idea of the things that are possible. In this column we leveraged the LightweightClasses and MethodProducers of our previous articles to implement ActiveVariables. We also showed another way to specialize the compilation process for a very particular need. We want to delve into the heart of the Smalltalk environment to help you understand it better and see possible ways to extend it for your own benefit. This month we spent most of our time discussing new code, but in the process we hope you learned a little about Behaviors and the compilation process. Finally, we are interested in hearing from you if you have comments or questions. Are we achieving our goals as far as you are concerned? Are there particular areas of the environment you'd like to understand, or advanced projects you'd like to see implemented? If you have any thoughts or feedback, please send them to Bob Hinkle by email at hinkle@primenet.com.

References

1. Messick, S. L. and K. L. Beck. "Active Variables in Smalltalk-80." *Technical Report CR-85-09*. Computer Research Lab, Tektronix, Inc. 1985.
2. Hinkle, B., V. Jones, and R. E. Johnson. "Debugging Objects." *The Smalltalk Report*, (2) 9, July-Aug. 1993.