Extending Advice Activation in AspectS

Robert Hirschfeld
DoCoMo Communications Laboratories Europe
Future Networking Lab
Landsberger Strasse 312
D-80687 Munich, Germany
hirschfeld@docomolab-euro.com

Pascal Costanza
Vrije Universiteit Brussel
Programming Technology Lab
Pleinlaan 2
B-1050 Brussel, Belgien
pascal.costanza@vub.ac.be

ABSTRACT
AspectS provides a mechanism called activation blocks as a means to control the invocation of advice code at instrumented join-point shadows. We describe all steps necessary to utilize this device. As an example, we exercise this recipe to make advice code process-aware, emphasizing the flexibility AspectS gained from taking a framework approach to AOP.

Keywords
AspectS, Dynamic Weaving, Runtime Composition, Reflection, Meta-programming, Context, Adaptability, Squeak.

1. INTRODUCTION
AspectS is intended to provide an AOP platform for dynamic environments that is both easy to use and simple to extend. It is easy to use due to a small framework kernel and no extensions made to the Smalltalk language. Extensions to AspectS are most of the time simple to carry out since many of the variation points needed to do so are either already there or straightforward to introduce.

Besides enabling developers to tailor their aspect compositions as needed, this approach gives also some degree of freedom in experimenting with new composition constructs. Advice qualifiers and their associated activation blocks are one mechanism allowing for that.

In the following we describe the steps necessary to add new compositional properties based on activation blocks. More specifically, we extend AspectS to support process-specific advice.

Since we found us to use this kind of extension frequently, we decided to document one such implementation that other developers wanting to modify AspectS in a similar manner could use as an initial reference to get started.

2. ASPECTS
AspectS is described in more detail in [3]. Here we provide a trivial example to summarize its basic pattern of use. Smalltalk provides an in-image console called Transcript for echoing messages sent via show: to the user. In the following we instrument this notification mechanism is such a way that every message to be displayed will be prefixed by the current time. First we create a new class named TimestampedTranscriptAspect as a subclass of AsAspect.

```
AsAspect subclass: #TimestampedTranscriptAspect
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AspectS-Examples TimestampedTranscript'
```

In this class, we implement the following advice method that, after installation, intercepts all show: messages sent to TranscriptStream (the actual implementation of Transcript), prints out the current time, and then passes on the control to show:'s original implementation. This behavior is mainly expressed in the before block that states what to execute at the locations (that is, message sends and receptions) designate by our pointcut enumeration.

```
TimestampedTranscriptAspect>>adviceTranscriptStreamShow
    ^ AsBeforeAfterAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. #cfFirstClass. })
        pointcut: [{
            AsJoinPointDescriptor
                targetClass: TranscriptStream
                targetSelector: #show:. }]
        beforeBlock: [:receiver :arguments :aspect :client |
            Transcript show: '[', Time now printString, '] ']
```

To restrict our composition further, we provide advice qualifier attributes. Since we do not want to change the behavior of a specific but all transcripts, we say so by providing the #receiverClassSpecific attribute.

Please note that in the before block we ourselves invoke the show: method that is going to be affected by our advice. To avoid infinite recursion once our aspect is installed, we provide an additional advice qualifier attribute, #cfFirstClass, that makes sure that our advice code is activated only if this is the first invocation of that kind in the current flow of control.

The following script illustrates the effect the installation of our aspect has.

```
doIt: [
    | assi |
    assi ← TimestampedTranscriptAspect new.
```

```
    3 timesRepeat: [Transcript cr; show: Smalltalk bytesLeft].
    assi install.
    3 timesRepeat: [Transcript cr; show: Smalltalk bytesLeft].
    assi uninstall.
    3 timesRepeat: [Transcript cr; show: Smalltalk bytesLeft].
]
```

In our script we first echo the number of bytes currently available in our image after a garbage collect to the Transcript. There. each line contains only the number of bytes left, but nothing else. After we install our TimestampedTranscriptAspect and print the same information, each line is prefixed with the current time. Once our aspect is uninstalled, the information to be displayed shows up without the aforementioned prefix, exhibiting behavior as of before of the aspect installation.

Transcript

```
1051276248
1051276356
1051276292
[9:41:44 pm] 1051274868
[9:41:44 pm] 1051274676
[9:41:44 pm] 1051274600
1051275576
1051275512
1051275448
```

3. ACTIVATION BLOCKS

In the previous section we explain how advice qualifiers and their attributes can be applied to affect the activation of advice code. Here we describe how advice qualifier attributes are taken into consideration for advice activation by the AspectS framework.

AspectS uses Method Wrappers [1] to instrument both message sends and receptions. Such wrappers let us execute additional code before, after, around, or instead of an existing method. Instead of modifying Smalltalk's standard lookup process, Method Wrappers change the objects this lookup process returns. Normally, each such lookup gives back a compiled method associated with the selector of the message received, or fails otherwise. Method Wrappers allow us to decorate compiled methods to realize the behavior outlined above.

To be used in AspectS, we extended Method Wrappers to be generic with respect to both the code to be executed and the conditions under which such code to be executed. We provided this degree of extensibility since it is impossible to anticipate in advance all forms of potential use.

The core of our activation mechanism is implemented in the following isActive method of AsMethodWrapper. All additional code provided by a wrapper is only to be active if all activation blocks associated with it evaluate to true. This is to say that activation blocks are treated as predicate methods, returning either true or false as the outcome of their execution.

```
AsMethodWrapper>>isActive
    | baseSender |
    baseSender ← thisContext baseSender.
    ^ self activators "<Set of: BlockContext>" notEmpty
        and: [self activators allSatisfy: [:aBlock |
            aBlock value: self aspect value: baseSender]]
```

Here is the simplest activation block possible. It returns always true and is used to express that the advice code introduced via the associated wrapper will be executed by all receivers of the message that are an instance of a specified class. The code rendered in gray color addresses Squeak-specific issues and is only provided for sake of completeness.

```
AsMethodWrapper class>>receiverClassSpecificActivator
    ^ [:aspect :baseSender |
        aspect ← baseSender ← nil.
        true] copy fixTemps
```

The following activation block allows for class-first cflow advice. Its activation test examines Smalltalk's activation stack for one ore more senders with the same class as that of the receiver. If there are no more senders with the same class as the receiver's, the activation block evaluates to true, otherwise to false.

```
AsMethodWrapper class>>cfFirstClassActivator
    ^ [:aspect :baseSender |
        | lastCfPoint allCfPoints result |
        lastCfPoint ← AsCFlowPoint
            object: baseSender receiver class
            selector: baseSender selector.
        allCfPoints ← thisContext allBaseClientsWithSelector
            collect: [:each |
                AsCFlowPoint
                    object: each key class
                    selector: each value].
        result ← (allCfPoints occurrencesOf: lastCfPoint) = 1.
        aspect ← baseSender ← lastCfPoint ← allCfPoints ← nil.
        result] copy fixTemps
```

To decouple developers from the actual implementation of activation blocks when they work on their aspect and advice code, we provide them with attributes such as #receiverClassSpecific and #cfFirstClass from our example that they can use to configure and advice qualifier.

Advice qualifier attributes can be associated with a corresponding activation block in initialize methods like the following two.

```
AsMethodWrapper class>>initRegularProtoActivators
    self protoActivators
        add: (#receiverClassSpecific
            -> self receiverClassSpecificActivator);
        add: (#receiverInstanceSpecific
            -> self receiverInstanceSpecificActivator);
        add: (#senderClassSpecific
            -> self senderClassSpecificActivator);
        add: (#senderInstanceSpecific
            -> self senderInstanceSpecificActivator).
```

The two initialize methods (initRegularProtoActivators and initCFlowProtoActivators) are called from within AsMethodWrapper class>>initialize.

```
AsMethodWrapper class>>initCFlowProtoActivators
    self protoActivators
        add: (#cfFirstClass
            -> self cfFirstClassActivator);
        add: (#cfAllButFirstClass
            -> self cfAllButFirstClassActivator);
        add: (#cfFirstInstance
            -> self cfFirstInstanceActivator);
        add: (#cfAllButFirstInstance
            -> self cfAllButFirstInstanceActivator);
        add: (#cfFirstSuper
            -> self cfFirstInstanceActivator);
        add: (#cfAllButFirstSuper
            -> self cfAllButFirstInstanceActivator).
```

4. PROCESS-SPECIFIC ADVICE

In older versions of AspectS there was no support for process-specific advice code [3]. Here we illustrate all steps that were necessary to implement for process-specific advice in AspectS.

4.1 An Example

We start by implementing the example from [2] (chapter Processes, p. 255), demonstrating multi-process behavior in Smalltalk. Please note that what is commonly called a thread in many other popular object-oriented programming languages such as Java is called a process in Squeak/Smalltalk [4].

The original example exhibits Smalltalk's priority and scheduling mechanism by starting multiple processes from a workspace and examining their traces left on the Transcript. Since we want to instrument parts of the code later on to demonstrate process-specific aspects more easily, we put that example code into a class. So, first we add a class ProcessExample with an instance variable named wordArray and all necessary accessing and convenience methods.

```
Object subclass: #ProcessExample
    instanceVariableNames: 'wordArray'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AspectS-Tests ProcessSpecific'

ProcessExample>>wordArray
    ^ wordArray

ProcessExample>>wordArray: anArray
    wordArray ← anArray.

ProcessExample>> wordArrayAt: anInteger put: aString
    self wordArray at: anInteger put: aString.

ProcessExample>>showWordArray
    Transcript cr; show: self wordArray.
```

Now we adapt and implement the above mentioned example from [2].

```
ProcessExample>>wordProcessExample
    | wordProcess |
    self wordArray: (Array new: 5 withAll: '---').
    wordProcess ← [
        [self wordArrayAt: 1 put: 'now'. self showWordArray]
            forkAt: Processor lowIOPriority "=> 60".
        [self wordArrayAt: 2 put: 'is'. self showWordArray]
            forkAt: Processor userInterruptPriority "=> 50".
        self wordArrayAt: 3 put: 'the'. self showWordArray]
            newProcess
                "activePriority => userSchedulingPriority => 40".
    wordProcess priority: Processor highIOPriority "=> 70".
    self wordArrayAt: 4 put: 'time'. self showWordArray.
    wordProcess resume.
    self wordArrayAt: 5 put: 'for'. self showWordArray.
```

Evaluating [ProcessExample new wordProcessExample] lets us observe the following messages on the Transcript. The result shows that the different processes populate different elements of the given word array in the order matching their relative scheduling priority.

```
                                                        Transcript
#('---'    '---'    '---'    'time'  '---')
#('---'    '---'    'the'    'time'  '---')
#('now'    '---'    'the'    'time'  '---')
#('now'    'is'     'the'    'time'  '---')
#('now'    'is'     'the'    'time'  'for')
```

4.2 Regular Instrumentation

We add an aspect ProcessExampleIndifferentAspect that is supposed to embed all words put in the wordArray within leading and trailing asterisks. Note that this aspect is not yet process-specific.

```
AsAspect subclass: #ProcessExampleIndifferentAspect
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AspectS-Tests ProcessSpecific'

ProcessExampleIndifferentAspect>> adviceWordArrayAtPut
    ^ AsAroundAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: [{ AsJoinPointDescriptor
            targetClass: ProcessExample
            targetSelector: #wordArrayAt:put:. }]
        aroundBlock: [:receiver :args :aspect :client :clientMethod |
            clientMethod
                valueWithReceiver: receiver
                arguments: { args first. '*** ', args second, ' ***'. }]
```

We extend our wordProcessExample into wordProcessExample2 where we make use of our previously developed ProcessExampleIndifferentAspect aspect.

```
ProcessExample>>wordProcessExample2
    | wordProcess processIndifferentAspect |
    processIndifferentAspect ←
        ProcessExampleIndifferentAspect new.
    processIndifferentAspect install.
    self wordArray: (Array new: 5 withAll: '---').
    wordProcess ← [
        [self wordArrayAt: 1 put: 'now'. self showWordArray]
            forkAt: Processor lowIOPriority "=> 60".
        [self wordArrayAt: 2 put: 'is'. self showWordArray]
            forkAt: Processor userInterruptPriority "=> 50".
        self wordArrayAt: 3 put: 'the'. self showWordArray]
            newProcess
                "activePriority => userSchedulingPriority => 40".
    wordProcess priority: Processor highIOPriority "=> 70".
    self wordArrayAt: 4 put: 'time'. self showWordArray.
    wordProcess resume.
    self wordArrayAt: 5 put: 'for'. self showWordArray.
    processIndifferentAspect uninstall.
```

Evaluating [ProcessExample new wordProcessExample2] lets us observe the following messages on the Transcript.

Transcript

#('---'	'---'	'---'	'*** time ***'	'---')
#('---'	'---'	'*** the ***'	'*** time ***'	'---')
#('*** now ***'	'---'	'*** the ***'	'*** time ***'	'---')
#('*** now ***'	'*** is ***'	'*** the ***'	'*** time ***'	'---')
#('*** now ***'	'*** is ***'	'*** the ***'	'*** time ***'	'*** for ***')

4.3 Projected Process-specific Advice

We now sketch an aspect implementation that shows how we would like the use of a process-specific aspect and its advice appear to a developer. For that, we just make a copy of ProcessExampleIndifferentAspect and name it ProcessExampleSpecificAspect.

```
AsAspect subclass: #ProcessExampleSpecificAspect
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AspectS-Tests ProcessSpecific'
```

We keep the advice method, but change the embedding characters from '***' to '###' to better see the difference in effect to the previously implemented process-indifferent aspect and its advice. We also make use of #processSpecific as an additional advice qualifier attribute to mark this advice as behavior in a process-specific way. Ideally, this is all that should be necessary to make an advice process-specific.

```
ProcessExampleSpecificAspect>>adviceWordArrayAtPut
    ^ AsAroundAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. #processSpecific. })
        pointcut: [{ AsJoinPointDescriptor
            targetClass: ProcessExample
            targetSelector: #wordArrayAt:put:. }]
        aroundBlock: [:receiver :args :aspect :client :clientMethod |
            clientMethod
                valueWithReceiver: receiver
                arguments: { args first. '### ', args second, ' ###'. }]
```

We also extend our wordProcessExample2 into wordProcessExample3 to now make use of our ProcessExampleSpecificAspect aspect.

```
ProcessExample>>wordProcessExample3
    | wordProcess processSpecificAspect |
    processSpecificAspect ←
        ProcessExampleSpecificAspect new.
    processSpecificAspect install.
    self wordArray: (Array new: 5 withAll: '---').
    wordProcess ← [
        [self wordArrayAt: 1 put: 'now'. self showWordArray]
            forkAt: Processor lowIOPriority "=> 60".
        [self wordArrayAt: 2 put: 'is'. self showWordArray]
            forkAt: Processor userInterruptPriority "=> 50".
        self wordArrayAt: 3 put: 'the'. self showWordArray]
            newProcess
                "activePriority => userSchedulingPriority => 40".
    wordProcess priority: Processor highIOPriority "=> 70".
    self wordArrayAt: 4 put: 'time'. self showWordArray.
    wordProcess resume.
    self wordArrayAt: 5 put: 'for'. self showWordArray.
    processSpecificAspect uninstall.
```

Note that the evaluation of [ProcessExample new wordProcessExample3] will fail since there is no process-specific behavior implemented yet – neither the #processSpecific advice qualifier attribute nor its associated behavior!

4.4 Making AspectS Process-aware

At this time we implement everything necessary to actually make aspects process-specific. While doing so, we want to allow process-specific aspects and advice to be associable with one or more processes at the same time.

We extend AsAspect with one more instance variable called processes. This new instance variable is used to recall all processes a particular aspect is associated with.[1]

```
Object subclass: #AsAspect
```

[1] While it would be an equally valid approach to implement our extensions in a subclass of AsAspect, we feel that since process-awareness is a very basic property and it needs to placed in the most appropriate base class of the AspectS, AsAspect itself.

```
    instanceVariableNames: 'receivers senders senderClasses
        projects processes clientAnnotations advice installed'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AspectS-Aspects'
```

We also provide all additional accessing, convenience, and initialization methods to set the list of processes as made necessary by both the AspectS framework as well as the client behavior.

```
AsAspect>>processes
    ^ processes

AsAspect>>processes: anIdentitySet
    processes ← anIdentitySet.

AsAspect>>initialize
    self
        receivers: IdentitySet new;
        senders: IdentitySet new;
        senderClasses: IdentitySet new;
        projects: IdentitySet new;
        processes: IdentitySet new;
        clientAnnotations: IdentityDictionary new;
        advice: nil;
        installed: false.

AsAspect>>addProcess: aProcess
    ^ self processes add: aProcess

AsAspect>>removeProcess: aProcess
    ^ self processes remove: aProcess ifAbsent: []

AsAspect>>hasProcess: aProcess
    ^ self processes includes: aProcess
```

We add a process-specific activator method to AsMethodWrapper class.

```
AsMethodWrapper class>>processSpecificActivator
    ^ [:aspect :baseSender |
        | result |
        result ← aspect hasProcess: Processor activeProcess.
        aspect ← baseSender ← nil.
        result] copy fixTemps
```

We make sure that the new activator is made known to the system by updating the appropriate initialization method, and subsequently initializing the list of activators with the new one.

```
AsMethodWrapper class>>initAdditionalProtoActivators
    self protoActivators
        add: (#projectSpecific -> self projectSpecificActivator);
        add: (#processSpecific -> self processSpecificActivator).
```

We can execute [AsMethodWrapper initialize] to make our changes effective. Now, we evaluate, once again, the code snipped that failed previously ([ProcessExample new wordProcessExample3]) and observe its printed trace on the Transcript.

Transcript
```
#('---'    '---'    '---'    'time'  '---')
#('---'    '---'    'the'    'time'  '---')
#('now'    '---'    'the'    'time'  '---')
#('now'    'is'     'the'    'time'  '---')
#('now'    'is'     'the'    'time'  'for')
```

Note that nothing has changed compared to the set-up with no aspects involved. This is because our new aspect is process-specific, but no process has been made known to our aspect, yet.

4.5 Applying Process-specific Advice

At this point, we continue to extend our wordProcessExample3 into wordProcessExample4 by making our aspect instance aware of the process responsible for printing out the first element of the word array (‘now’).

```
ProcessExample>>wordProcessExample4
    | wordProcess processSpecificAspect |
    processSpecificAspect ← ProcessExampleSpecificAspect new.
    processSpecificAspect install.
    self wordArray: (Array new: 5 withAll: '---').
    wordProcess ← [
        [processSpecificAspect
            addProcess: Processor activeProcess.
        self wordArrayAt: 1 put: 'now'. self showWordArray]
            forkAt: Processor lowIOPriority "=> 60".
        [self wordArrayAt: 2 put: 'is'. self showWordArray]
            forkAt: Processor userInterruptPriority "=> 50".
        self wordArrayAt: 3 put: 'the'. self showWordArray]
            newProcess
                "activePriority => userSchedulingPriority => 40".
    wordProcess priority: Processor highIOPriority "=> 70".
    self wordArrayAt: 4 put: 'time'. self showWordArray.
    wordProcess resume.
    self wordArrayAt: 5 put: 'for'. self showWordArray.
    processSpecificAspect uninstall.
```

The evaluation of [ProcessExample new wordProcessExample4] yields to the desired trace of print-outs.

Transcript
```
#('---'                '---'    '---'    'time'  '---')
#('---'                '---'    'the'    'time'  '---')
#('### now ###'        '---'    'the'    'time'  '---')
#('### now ###'        'is'     'the'    'time'  '---')
#('### now ###'        'is'     'the'    'time'  'for')
```

4.6 Validating the Desired Behavior

Finally we verify that a process-specific aspect can be associated with more than one process by adding the process the process-aware aspect was created in to list of process our aspect is to be aware of. We do so in method wordProcessExample5.

```
ProcessExample>>wordProcessExample5
    | wordProcess processSpecificAspect |
    processSpecificAspect ←
        ProcessExampleSpecificAspect new.
    processSpecificAspect
        addProcess: Processor activeProcess.
    processSpecificAspect install.
    self wordArray: (Array new: 5 withAll: '---').
    wordProcess ← [
        [processSpecificAspect
            addProcess: Processor activeProcess.
        self wordArrayAt: 1 put: 'now'. self showWordArray]
            forkAt: Processor lowIOPriority "=> 60".
        [self wordArrayAt: 2 put: 'is'. self showWordArray]
            forkAt: Processor userInterruptPriority "=> 50".
        self wordArrayAt: 3 put: 'the'. self showWordArray]
            newProcess
                "activePriority => userSchedulingPriority => 40".
    wordProcess priority: Processor highIOPriority "=> 70".
    self wordArrayAt: 4 put: 'time'. self showWordArray.
    wordProcess resume.
    self wordArrayAt: 5 put: 'for'. self showWordArray.
    processSpecificAspect uninstall.
```

The evaluation of [ProcessExample new wordProcessExample5] leads now to the following trace that meets our expectations.

```
                                                        Transcript
#('---'          '---'   '---'   '### time ###'   '---')
#('---'          '---'   'the'   '### time ###'   '---')
#('### now ###'  '---'   'the'   '### time ###'   '---')
#('### now ###'  'is'    'the'   '### time ###'   '---')
#('### now ###'  'is'    'the'   '### time ###'   '### for ###')
```

5. SUMMARY AND FUTURE WORK

In this paper we explain how dynamic composition properties of AspectS can be extended by implementing new advice qualifier attributes and their associated activation blocks. We demonstrate all steps necessary to do so using process-specific advice as an example. This implementation is now part of the AspectS release [5]. With our approach we show the great flexibility gained from a fully dynamic reflective environment in general and a framework approach to AOP specifically.

Next steps could involve the provisioning of convenience methods that let us register new advice qualifiers and their associated activation blocks more dynamically (including the initialization or update of protoActivators in AsMethodWrapper) – Something like that: ☺

```
AsMethodWrapper class>>addQualifier: aSymbol activationBlock:
aBlockContext
    self protoActivators add: aSymbol -> aBlockContext.
```


6. ACKNOWLEDGMENTS

We want to thank Michael Haupt and Stefan Hanenberg for their valuable comments and suggestions.

7. REFERENCES

[1] J. Brant, B. Foote, R. Johnson, D. Roberts. *Wrappers to the Rescue.* In Proceedings of the 1998 European Conference on Object-Oriented Programming (ECOOP), pp. 396-417, Brussels, Belgium, 1998

[2] A. Goldberg, D. Robson. *Smalltalk 80 – The Language and its Implementation.* Addison-Wesley, 1983 (pp. 254-257).

[3] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In *M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World,* pp. 216-232, LNCS 2591, Springer, 2003.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA),* pp. 318-326, ACM Press, Atlanta, GA, USA, October 5-9, 1997.

[5] AspectS homepage for download (http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/).