Advice Activation in AspectS

Robert Hirschfeld
DoCoMo Euro-Labs
hirschfeld@docomolab-euro.com
http://www.docomolab-euro.com
January 2002

Abstract

AspectS supports aspect-oriented programming (AOP) in Smalltalk using meta-programming. The goal of the AspectS project is to provide a platform for the exploration of AOP in the context of dynamic systems. In its implementation, AspectS makes use of Method Wrappers as an approach to place advice-related computation at specific join points. Different kinds of Method Wrappers are used for different kinds of advice. One of the key mechanisms employed by these Method Wrappers, the way by which they are activated at runtime, is described in this paper.

1. Introduction

Aspect-Oriented Programming is based on the assumption that crosscutting is inherent to complex systems [Kicz+97]. It addresses these issues by introducing new units of modularity to capture crosscutting structures explicitly. Such structures are called aspects and can be found in a software system's design as well as its implementation.

AspectS[1] is intended to be used as a tool to explore AOP from the perspective of dynamic software systems. Based on the language model of AspectJ, AspectS extends the Smalltalk[2] environment to accommodate the aspect modularity mechanism to allow for experimental aspect-oriented system development [Hirs01b]. It draws on the results of two projects: the first is AspectJ from Xerox PARC, a general-purpose aspect-oriented language extension to Java [AJ01, Kicz+01], and the second is John Brant's Method Wrappers, a powerful mechanism to add behavior to a compiled Smalltalk method [BFJR99, MW01].

AspectS introduces activation blocks to control the activation/deactivation of method wrappers at runtime, depending on the activation context. The following sections will illustrate the advice activation mechanism in the AspectS system, but not the AspectS system itself. A more general description of AspectS can be found in [Hirs01b]. It is assumed that the reader is familiar with Smalltalk [GoRo83].

2. Method Wrappers

Method Wrappers allow the introduction of code that is executed before, after, or instead of an existing compiled method.

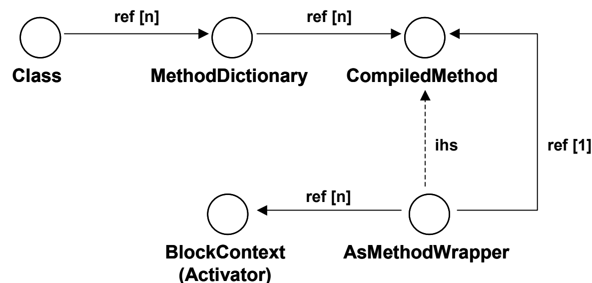

Figure 1[3]

[1] The version of AspectS discussed in the text is 0.2.2 (2001-10-24, [Hirs01a]).

[2] AspectS is implemented in Squeak and VisualWorks [Sque01, VW01]. Squeak is an open, highly portable Smalltalk-80 implementation. Its virtual machine is written entirely in Smalltalk [GoRo83]. VisualWorks is an industrial-strength Smalltalk, based on Smalltalk-80 as well.

[3] Squeak does not yet support the concept of namespaces. The AspectS system addresses this issue by a naming convention to prefix all of its classes with 'As'. The graphical notation used in this text is based on OOSE/Objectory [JCJÖ93].

As an alternative to modifying Smalltalk's standard lookup process, Method Wrappers change the method objects this lookup mechanism returns. A Method Wrapper replaces an entry in a class' method dictionary (a compiled method or a Method Wrapper), adds behavior to the method invocation, and may eventually invoke the wrapped method if wanted. The wrapped method itself can be either a compiled method or yet another Method Wrapper (Figure 1). Method Wrappers are described in more detail in [BFJR99].

In AspectS, each Method Wrapper is associated with an advice that belongs to an aspect. Depending on the advice qualifiers an advice object is configured with, Method Wrappers need to be active/inactive under different conditions at runtime. Such runtime conditions encompass, for example, the classes or particular instances of senders or receivers of a message, or control flow (cflow) semantics like the first or all but the first method invocation in an object or a method recursion.

3. Activation Blocks

AspectS' weaving process coordinates the positioning of Method Wrappers according to advice properties of the aspect to be installed. Each join point description denotes a weaving target by naming a class and one of its methods. The weaver modifies the class' method dictionary via the insertion of a proper Method Wrapper (Figure 2). In case of an introduction the wrapper wraps nothing (literally), otherwise it refers to a client method that can be either a regular compiled method or yet another Method Wrapper.

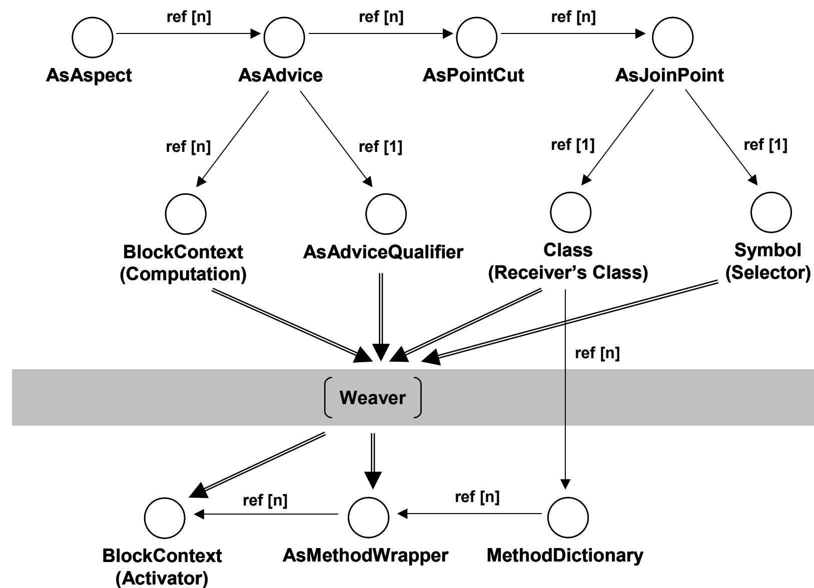

Figure 2

According to the attributes stated in an advice qualifier[4], a Method Wrapper is configured with one or more activation blocks. Each activation block, represented by a Smalltalk block, is provided with the aspect instance associated with the wrapper, and the base level activation context (base sender)[5] that allows access to not only the receiver of the message, but to the whole chain of activation contexts (Smalltalk's stack). Depending on this information, the activation block evaluates to a Boolean value, either true or false.

In the current implementation, Method Wrappers combine the results of all activation blocks via the Boolean AND operator, meaning that all activation blocks have to evaluate to true to put the wrapper into an active state:

```
AsMethodWrapper>>isActive

        | baseSender |
        baseSender := thisContext baseSender.
        ^ self activators notEmpty
                and: [self activators allSatisfy: [:aBlock |
                        aBlock value: self aspect value: baseSender]]
```

[4] Advice qualifier attributes can be compared to AspectJ's concept of pointcut designators.

[5] A base level activation context is an activation context who's receiver is neither a Method Wrapper nor a block context.

While an inactive wrappers passes execution control on to its client method (if any), an active wrapper is allowed to execute additional code before, after, or instead of its client method (if any).

The following examples will illustrate some of the activation blocks currently in use in AspectS. A receiver-general activation block always returns true since it states that the wrapper's computation can be carried out for all instances of the receiver's class:

AsMethodWrapper class>>receiverGeneralActivator

```
^ [:aspect :baseSender |
        true] copy
```

A receiver-specific activation block determines the actual receiver of the message and checks if this receiver was registered with the aspect instance:

AsMethodWrapper class>>receiverSpecificActivator

```
^ [:aspect :baseSender |
        aspect hasReceiver: baseSender receiver] copy
```

A cflow-first-class activation block examines the base context chain for one or more senders that belong to the same class as the receiver. Activation happens if there is only one such class in the context chain, meaning that the receiver is the first instance of its class in the call sequence:

AsMethodWrapper class>>cfFirstClassActivator

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

The following method shows the usage of the activation test by a before-after wrapper, a specialized Method Wrapper, that can perform additional code right before and right after a method invocation:

AsBeforeAfterWrapper>>valueWithReceiver: anObject arguments: anArrayOfObjects

```
| client active return |
client := thisContext baseClient.
active := self isActive.
active ifTrue: [self beforeBlock copy valueWithArguments: (Array
        with: anObject
        with: anArrayOfObjects
        with: self aspect
        with: client)].
return := self clientMethod
        valueWithReceiver: anObject
        arguments: anArrayOfObjects.
active ifTrue: [self afterBlock copy valueWithArguments: (Array
        with: anObject
        with: anArrayOfObjects
        with: self aspect
        with: client
        with: return)].
^ return
```

For more details, the reader is referred to the actual AspectS implementation [Hirs01a].

4. Final Remarks

With their pluggability, activation blocks provide flexibility in addressing the activation/deactivation of Method Wrappers in the AspectS environment. In a future release, the combination of activation results will be extended to a minimal functionally complete system of logic operators (like AND, OR, and NOT).

References

[AJ01] *AspectJ homepage* (http://aspectj.org)

[BFJR99] Brant, John; Foote, Brian; Johnson, Ralph; Roberts, Don: *Wrappers to the Rescue*. In: ECOOP'98 Proceedings, 1998

[GoRo83] Goldberg, Adele; Robson, David: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983

[Hirs01a] Hirschfeld, Robert: *AspectS homepage* (http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/)

[Hirs01b] Hirschfeld, Robert: *AspectS – Aspects in Squeak*. Submitted to AOSD 2002

[JCJÖ93] Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1993

[Kicz+97] Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina Videira; Loingtier, Jean-Marc; Irwin, John: *Aspect-Oriented Programming*. In: ECOOP' 97 Proceedings, 1997

[Kicz+01] Kiczales, Gregor; Hilsdale, Erik; Hugunin, Jim; Kersten, Mik; Palm, Jeffrey; Griswold, William G.: *An Overview of AspectJ*. In: ECOOP' 01 Proceedings, 2001

[MW01] *MethodWrappers homepage* (http://st-www.cs.uiuc.edu/~brant/Applications/MethodWrappers.html)

[Sque01] *Squeak homepage* (http://squeak.org)

[VW01] *VisualWorks homepage* (http://www.parcplace.com, http://www.cincom.com)