Aspect-Oriented Programming with AspectS

Robert Hirschfeld

DoCoMo Communications Laboratories Europe
Landsberger Strasse 308-312, 80687 Munich, Germany

hirschfeld@docomolab-euro.com

Abstract. AspectS is an approach to general-purpose aspect-oriented programming in the Squeak/Smalltalk environment. Based on concepts of AspectJ it extends the Smalltalk metaobject protocol to accommodate the aspect modularity mechanism. In contrast to systems like AspectJ, weaving and unweaving in AspectS happens dynamically at runtime, on-demand, employing metaobject composition. Instead of introducing new language constructs, AspectS utilizes the expressiveness of Smalltalk itself as a pointcut language.

1 Introduction

Aspect-Oriented Programming (AOP) is based on the assumption that crosscutting is inherent to complex systems [12]. 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. As of today there are several approaches that support aspect-oriented concepts, ranging from general-purpose aspect languages like AspectJ [1, 13] to domain-specific aspect languages such as RG or D [15, 14].

AspectS[1] extends the Squeak/Smalltalk[2] environment to allow for experimental aspect-oriented system development [8]. It mainly draws on the results of two projects: the first is AspectJ from Xerox PARC, a general-purpose aspect-oriented language extension to Java, and the second is John Brant's MethodWrappers, a powerful mechanism to add behavior to a compiled Smalltalk method [2, 16]. It benefits greatly from the simple, elegant, and open architecture of Squeak itself as well [6, 9, 17].

[1] The version of AspectS discussed in the text is 0.4.1 (2002-04-03, [8]).

[2] AspectS is implemented in Squeak. Squeak is an open, highly portable Smalltalk-80 implementation [17]. Its virtual machine is written entirely in Smalltalk. The terms Squeak and Smalltalk are used interchangeably in this text. There is also a port of AspectS to VisualWorks, another Smalltalk-80 derivative [18].

The goal of AspectS is to provide a platform for the exploration of aspect-oriented software composition in the context of dynamic systems. It supports coordinated meta-level programming, addressing the tangled code phenomenon by providing aspect related modules. In its current implementation, AspectS is realized without changing neither Smalltalk's syntax nor its virtual machine. AspectS shows great flexibility by not relying on code transformations (neither source nor byte code) but making use of metaobject composition instead.

The remainder of the paper is organized as follows: Sections 2, 3, and 4 introduce AspectS' implementation of aspects, join points, and advice objects. Section 5 gives an overview of the dynamic on-demand weaving mechanism employed by AspectS. Selected examples are discussed in section 6. Section 7 illustrates basic tool support. Section 8 puts AspectS in context with other AOP implementations for Smalltalk. A summary and final remarks are given in section 9. All of the ideas described in the paper are fully implemented by the author.

2 Aspects

Aspects (AsAspect[3]) are units of modularity that represent implementations of crosscutting concerns. Aspects associate code fragments (code to be executed when a join point is encountered) with join points (well-defined spots in the execution of code) by the use of advice objects (AsAdvice). A collection of related join points, to be addressed by an advice, is called a pointcut (Figure 1)[4].

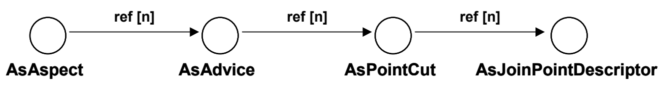

Figure 1

In AspectS, aspects are implemented via regular classes, so their instances act as regular objects also. An aspect is applied to objects in the image by sending an install message to an aspect instance. The effects of an aspect to the system are reverted by simply sending an uninstall message to the same aspect instance that cause the system transformation.

Since AspectS uses a framework-based approach to AOP, the described object model is directly reflected in the code one writes to express aspects, as illustrated later by examples in section 6.

[3] Class names in AspectS are prefixed with 'As' to avoid collisions with names of other classes since Squeak, in its current implementation, supports only a global namespace.

[4] The notation used in this text is based on OOSE/Objectory [10].

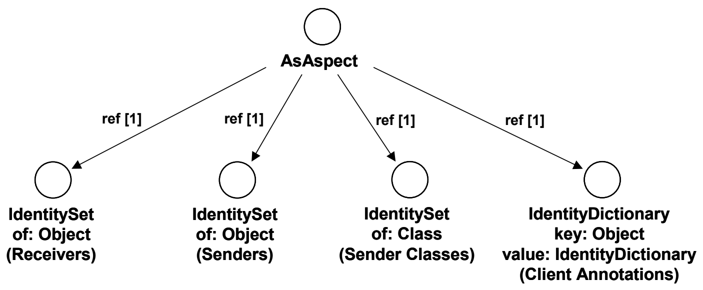

Figure 2

An aspect may hold on to a set of receivers, senders, or sender classes (Figure 2). These objects are added or removed by client code, and will be used by woven (composed) code at run-time to determine if receiver-instance-specific, sender-instance-specific or sender-class-specific behavior has to be activated or not. Client annotations allow the introduction of advice-specific state.

3 Join Points

Join points are well-defined spots in the execution of code. Join point descriptors (AsJoinPointDescriptor) denote targets for the weaving process to apply computational changes to the underlying base system stated in advice objects.

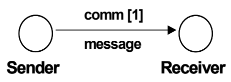

Figure 3

In Smalltalk, object interaction is based on the message-sending metaphor (Figure 3). A message sent by a sender is decoupled from the actual method implementation executed by the receiver on behalf of the sender. In AspectS the receiver of a message is considered the only structural information related to a join point. By naming a target class and a target selector, a join point descriptor partially describes a join point's static property of location within the system's class hierarchy (Figure 4). Advice qualifiers (discussed in section 4.2) allow the description of dynamic attributes of a join point in the context of an advice.

Join points of a point cut can be enumerated statically, or, due to the very open and reflective nature of the Smalltalk environment, collected dynamically by querying the system. AspectS does not introduce a dedicated pointcut language but takes advantage of the expressiveness of Smalltalk itself

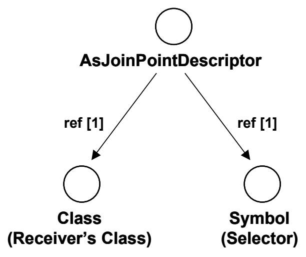

Figure 4

Note that the description of a join point combining a target class and a target selector is not polymorphic but strict. Polymorphic characteristics can be expressed via dynamic system queries for join point collection as described above.

4 Advice

Advice objects associate code fragments (parts of the crosscutting concern to be implemented by an aspect) with pointcuts and their respective join points descriptors that describe targets for the weaver to place these fragments into the system.

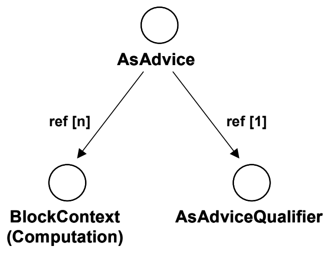

Figure 5

AspectS uses blocks (instances of BlockContext) to represent code fragments (Figure 5). An advice is to be qualified to state if the woven code will be receiver-class-aware, receiver-instance-aware, sender-class-aware or sender-instance-aware, combined with additional cflow (call flow) semantics if needed.

4.1 Kinds of Advice

AspectS, in its current version, allows to execute crosscutting behavior (Figure 6):

- before and after the execution of a method invocation (AsBeforeAfterAdvice),
- to handle signaled exceptions (AsHandlerAdvice), and
- around a method invocation (AsAroundAdvice)

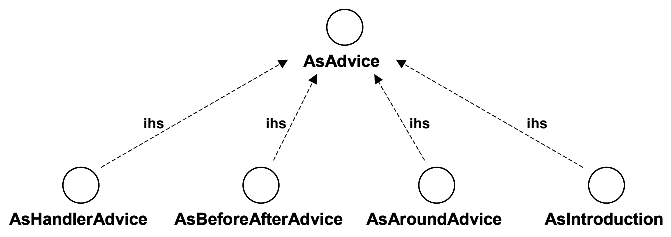

Figure 6

It is possible to *introduce* new behavior to the target clients as well (AsIntroduction). Table 1 gives an overview of the different kinds of advice:

Table 1

Kind of Advice	Description
Handler	A handler advice (AsHandlerAdvice) allows to place an exception handler around a message send. It specifies an exception class and an associated exception handler block that is to be executed if the message send results in the signaling such an exception. The additional code, represented as a block context, has access to all method parameters as well as the exception raised.
Before-After	With a before-after advice (AsBeforeAfterAdvice) one can perform additional code right before and right after a method invocation. The additional code, represented as block contexts also, has access to all method parameters.
Around	An around advice (AsAroundAdvice) may be put in front of a specific method to explicitly control the activation of that method with respect to the actual execution context. The additional code, represented as a block context, has access to all method parameters.
Introduction	With an introduction (AsIntroduction) one can introduce new behavior that is needed in the aspect's context. The added behavior may be invoked by the aspect, and may actively invoke the aspect's or client's behavior itself.

These kinds of advice are not minimal since both a handler advice and a before-after advice can be expressed using an around advice. However, providing these concepts makes the intent of some advice objects more obvious and so the resulting code better to understand.

4.2 Advice Qualifier

An advice qualifier (AsAdviceQualifier) allows the description of dynamic attributes of a pointcut related to an advice. These attributes state dynamic characteristics of join points enumerated by a pointcut in the context of an advice.

Advice qualifier attributes can be grouped roughly into:

- sender/receiver aware activation, and
- cflow (control or call flow) activation.

The combination of point cut descriptors and advice qualifier attributes can be compared to AspectJ's concept of pointcut designators.

Receiver/Sender-aware Activation

Table 2 gives an overview of sender and receiver specific advice qualifier attributes. Note that Receiver/Sender and Class/Instance-Specific are orthogonal.

Table 2

Qualifier Attribute	Description
Receiver Class Specific	With a receiver-class-specific advice, all receivers of the message that are an instance of a certain class are going to be affected. The advice is receiver-class-aware.
Receiver Instance Specific	With a receiver-instance-specific advice, only specific receivers of the message that are an instance of a certain class are going to be affected. Instances of prospective receivers can be added to or removed from the advice's aspect. The advice is receiver-instance-aware.
Sender Class Specific	With a sender-class-specific advice, receivers of the message that are an instance of a certain class are going to be affected if the sender is of a certain class or its subclasses. Sender classes can be added to or removed from the advice's aspect. The advice is sender-class-aware.
Sender Instance Specific	With a sender-instance-specific advice, receivers of the message that are an instance of a certain class are going to be affected only if the sender is known to the advice. Instances of prospective senders can be added to or removed from the advice's aspect. The advice is sender-instance-aware.

Cflow

Table 3 gives an overview of cflow related advice qualifier attributes. Note that Class/Instance and First/All-But-First are orthogonal.

Table 3

Qualifier Attribute	Description
Class First	With a class-first cflow advice, the activation test examines the base context chain (Smalltalk's stack) for one or more senders with the same class as the receiver's. Activation happens if there is only one such class in the context chain. (Example: Such advice will trigger activation on an object-recursion's first method invocation.)
Class All-But-First	With a class-all-but-first cflow advice, the activation test examines the base context chain for one or more senders with the same class as the receiver's. Activation happens if there is more than one such class in the context chain. (Example: Such advice will trigger activation on an object-recursion's other than first method invocation.)
Instance First	With an instance-first cflow advice, the activation test examines the base context chain for one or more appearances of the receiver instance in it. Activation happens if there is only one such instance in the context chain. (Example: Such advice will trigger activation on a method-recursion's first method invocation.)
Instance All-But-First	With an instance-all-but-first cflow advice, the activation test examines the base context chain for one or more appearances of the receiver instance in it. Activation happens if there is more than one such instance in the context chain. (Example: Such advice will trigger activation on a method-recursion's other than first method invocation.)
Super First	With a super-first cflow advice, the activation test examines the base context chain for a send of the current message to super. Activation happens if there was no send of the current message to super.
Super All-But-First	With a super-first cflow advice, the activation test examines the base context chain for a send of the current message to super. Activation happens if there was a send of the current message to super.

5 On-Demand Weaving

The activity of integrating aspects and their advice into the base system is called weaving. Weaving in general can be performed at compile-time or run-time. AspectJ is an example for compile-time weaving. Here, the weaver parses an AspectJ program, transform the AspectJ abstract syntax tree (AST) into a valid Java AST, and then generates Java byte code for a standard Java virtual machine. JMangler performs load-time transformation of Java class files [11]. AspectS employs a run-time weaver to transform the base system according to the aspects involved. The woven code is based on method wrappers and meta-programming.

5.1 Method Wrappers

Method wrappers allow for the introduction of code that is executed before, after, or instead of an existing method. As an alternative to modifying Smalltalk's standard lookup process, method wrappers change the objects the lookup mechanism returns.

A method wrapper replaces an entry in a class' method dictionary (a compiled method or another method wrapper), adds behavior to the method invocation, and eventually invokes the wrapped method itself (Figure 7).

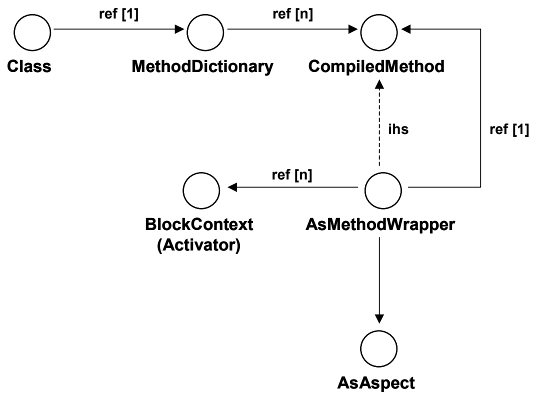

Figure 7

AspectS makes use of block method wrappers, special wrappers that allow to plug-in block contexts for additional behavior. For each kind of advice (Figure 6) there is a matching method wrapper implementation (Figure 8).

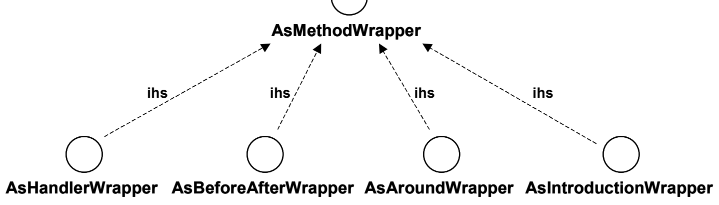

Figure 8

5.2 Weaving Aspects

AspectS coordinates the placement of block method wrappers into the method dictionaries of the classes of the receivers stated in the various join points advised by the aspect (Figure 9).

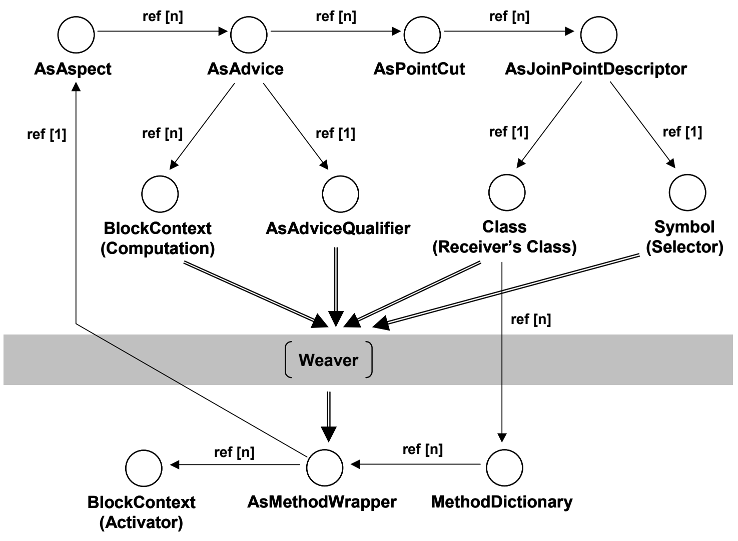

Figure 9

The weaving process happens every time an aspect instance is installed by sending an install message to the respective aspect instance. To reverse the effects of an aspect to the system, the aspect hast to be uninstalled by sending an uninstall message to the aspect instance responsible for the system transformation to be reversed. This process is also referred to as unweaving. Weaving and unweaving in AspectS can be characterized as completely dynamic since it happens at runtime. AspectJ, in contrast, preprocesses code outside of the actual system in a separate preprocess or compile step.

Method wrappers are placed around a compiled method in such a way that their activation will happen in the following order, considering the type of the wrapper as well as the hierarchy of their originating aspects:

- Around advice/wrappers (most-specific first)
  - Before parts of before-after advice/wrappers (most-specific first)
    - Handler advice/wrappers activation (most-specific first)
      - Compiled method (base computation)
    - Handler advice/wrappers (least-specific first)
  - After parts of before-after advice/wrappers (least-specific first)
- Around advice/wrappers (least-specific first)

An advice is more specific than another if it is defined in an aspect that is more specific than the aspect the other advice is defined it. If two pieces of advice are either defined in the same aspect, or if their aspects are not related to each other via a direct or transitive inheritance relationship, the specificity between them is undefined. The complexity of the computation of an advice's 'most/least-specific' property dependent linearly on the number of wrappers installed at the specific join point and the depth of the aspect class hierarchy. This computation is performed only once per wrapper during the installation of its aspect, not during actual message sends themselves.

5.3 Advice Activation

According to the attributes stated in an advice qualifier, 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]]
```

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-class-specific 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>>receiverClassSpecificActivator
    ^ [:aspect :baseSender | true] copy[6]
```

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

[6] One of the issues with Squeak's blocks is that no new evaluation structures are created for their evaluation, and with that the parallel evaluation of the same block would lead to a

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

```
AsMethodWrapper class>>receiverInstanceSpecificActivator
    ^ [: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 [8].

runtime error indicating the attempt to evaluate a block that is already being evaluated. Copying the block avoids that situation.

6 Examples

The following examples demonstrate basic mechanisms of AspectS. In the first example, selected message sends are reported to the transcript. In the second example, aspects are used to instrument recursive calls.

6.1 Tracing

In the first example, the goal is to monitor all mouseEnter: and mouseLeave: messages sent to instances of Morph and its subclasses by logging them to the system transcript. In a plain Squeak image (version 3.0) there are 23 implementers of mouseEnter: and 20 implementers of mouseLeave. 22 of the 23 of mouseEnter: methods and 19 of the 20 of mouseLeave: methods are found in Morph and its subclasses. One would have to put the same code into 41 different places. If this code changes, it has to do so in all of the 41 locations.

Without aspects or any other type of meta-programming, the solution might look like this: One determines all the implementers of mouseEnter: and mouseLeave: methods, selects the ones that are Morph or its subclasses, and then modifies the method implementations to report every message reception. This very manual procedure affects many parts of the system in an uncoordinated way. As a result, code is duplicated and tangled all over the image instead of being stated once in a single location. Depending on the rate of change, keeping all adjustments in sync might become a challenge.

Aspects are a convenient way to address this challenge. An aspect called AsMorphicMousingAspect traces the reception of mouseEnter: and mouseLeave: messages by instances of Morph and its subclasses. All aspects are subclasses of AsAspect, and so is AsMorphicMousingAspect.

Advice to trace the reception of mouseEnter: and mouseLeave: messages are stated in two advice methods. The convention here is that an advice method's selector matches 'advice#*'[7] with no arguments, and that the method returns an instance of AsAdvice or one of its subclasses. Here, an AsBeforeAfterAdvice object is created that allows to set behavior before and after the invocation of a method. Once the advice object is created, it is further qualified via #receiverGeneral to execute the additional computation for all receivers described by the specified join points:

[7] 'advice#*' describes a pattern that matches with strings that start with 'advice' and contain at least one more character.

```
adviceMouseEnter
    ^ AsBeforeAfterAdvice new
            qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
            pointcut: [
                    Morph withAllSubclasses
                            select: [:each | each includesSelector: #mouseEnter:]
                            thenCollect: [:each | AsJoinPointDescriptor
                                    targetClass: each
                                    targetSelector: #mouseEnter:]]
            beforeBlock: [:receiver :arguments :aspect :client |
                    self
                            showHeader: '>>> MouseENTER >>>'
                            receiver: receiver
                            event: arguments first]
```

In adviceMouseEnter join points are collected by querying the system for all classes that are subclasses of Morph and implement mouseEnter:. The block to be executed before the actual invocation of mouseEnter: sends a message to the aspect itself to echo the receiver of the mouseEnter: message and its event parameter to the Transcript. An adviceMouseLeave advice works likewise for the reception of mouseLeave: messages. showHeader:receiver:event: performs the actual printout to the Transcript. Note that both the pointcut selection and the advice code can be factored out into separate methods to be called from within this or other advice methods and with that reusable elsewhere.

To activate the AsMorphicMousingAspect, one creates an aspect instance sends it an install message. The send of an uninstall message deactivates the aspect. Activation and deactivation is allowed to happen on-demand at any time during runtime.

6.2 Cflow: Object and Method Recursion

In this simple example, aspects are going to be used to instrument recursive calls. Cflow directing advice is applied to visualize object and method recursion in the calculation of factorials. While for object recursion the receiver of the message sent recursively is a different object, for method recursion the message is sent recursively to the same object. One implementation of factorial is that of Integer:

```
factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'.
```

This implementation uses object recursion for its calculation since every subsequent factorial message is sent to a different instance of Integer. To echo the initial reception of a factorial message to the Transcript, leaving out all internal invocations to calculate the factorial of the next smaller instance of Integer (Figure 10), adviceFactorialInFirst is applied. For that, a #cfFirstClass attribute has to be provided:

```
adviceFactorialInFirst
   ^ AsBeforeAfterAdvice
          qualifier: (AsAdviceQualifier
                 attributes: { #receiverClassSpecific. #cfFirstClass. })
          pointcut: [OrderedCollection
                 with: (AsJoinPointDescriptor
                        targetClass: Integer
                        targetSelector: #factorial)]
          beforeBlock: [:receiver :arguments :aspect :client |
                 | msg |
                 msg := String cr, '#factorial-in: ', receiver printString.
                 self traceFirst: self traceFirst, msg.
                 Transcript show: msg]
```

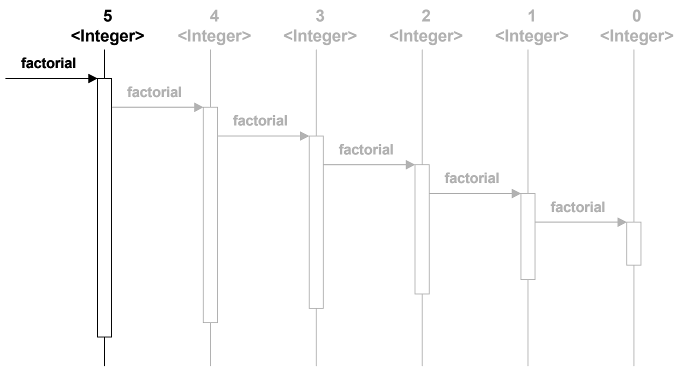

Figure 10

To report all but the initial reception of a factorial message in such scenario, an advice like adviceFactorialInAllButFirst can be utilized. Here, a #cfAllButFirstClass advice qualifier attribute has to be set:

```
adviceFactorialInAllButFirst
   ^ AsBeforeAfterAdvice new
          qualifier: (AsAdviceQualifier
                 attributes: { #receiverClassSpecific. #cfAllButFirstClass. })
          pointcut: [OrderedCollection
                 with: (AsJoinPointDescriptor
                        targetClass: Integer
                        targetSelector: #factorial)]
          beforeBlock: [:receiver :arguments :aspect :client |
                 | msg |
                 msg := String cr, '*factorial-in: ', receiver printString.
                 self traceAllButFirst: self traceAllButFirst, msg.
                 Transcript show: msg]
```

AsFactorial2 implements the calculation of factorials differently using method recursion by sending every subsequent factorial: message to itself:

```
factorial: anInteger
    anInteger = 0 ifTrue: [^ 1].
    anInteger > 0 ifTrue: [^ anInteger * (self factorial: anInteger - 1)].
    self error: 'Not valid for negative integers.'.
```

adviceFactorialInFirst applied will echo the initial reception of a factorial: message by a particular instance to the Transcript, leaving out all subsequent sends of factorial: to self (Figure 11):

```
adviceFactorialInFirst
    ^ AsBeforeAfterAdvice new
            qualifier: (AsAdviceQualifier
                    attributes: { #receiverClassSpecific. #cfFirstInstance. })
            pointcut: [OrderedCollection
                    with: (AsJoinPointDescriptor
                            targetClass: AsFactorial2
                            targetSelector: #factorial:)]
            beforeBlock: [:receiver :arguments :aspect :client |
                    | msg |
                    msg := String cr, '#factorial2-in: ', arguments first printString.
                    self traceFirst: self traceFirst, msg.
                    Transcript show: msg]
```

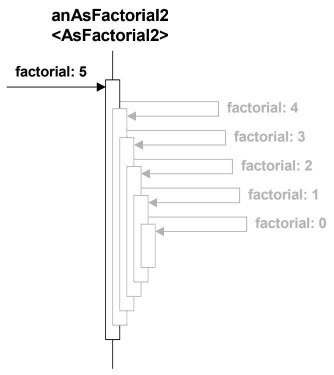

Figure 11

7 Tool Support

Most of the code browsers of the Squeak environment were extended to support aspect-oriented programming, both via regular object-oriented techniques and by the use aspects. AspectS browser extensions assist in bidirectional navigation from classes and methods affected by aspects and advice code. Parts of the system that were affected by the installation of one or more aspects such as methods with their method categories and classes with their class categories are highlighted in browsers. Menus of class and method lists were enhanced to allow for access to aspect classes as well as their specific instances that originated the change of the classes and methods highlighted in the browsers. Starting from an aspect class or one of its advice methods, one can explore pointcuts affected by the aspect or the particular advice.

8 Related Systems

AspectS is based on some of the ideas behind AspectJ. While AspectJ provides a language extension to Java to express AOP concepts, AspectS modifies the Smalltalk metaobject protocol (MOP). In contrast to AspectJ, weaving and unweaving in AspectS happens dynamically at run-time. While AspectJ's weaver performs code transformation at compile-time, the AspectS weaver is based on metaobject composition.

Dynamic Cool demonstrates the use of the Smalltalk MOP for the implementation of an AOP system [4]. Interesting efforts to offer AOP support for Smalltalk environments are AOP/ST for VisualWorks by Kai Böllert, Apostle for VisualAge for Smalltalk by Brian de Alwis, and Andrew for Squeak by Kris Gybels [3, 5, 7]. AOP/ST provides a programming environment for developing aspect modules to synchronize processes and to trace messages. Apostle aims to implement AspectJ in Smalltalk with special language support for defining join points, pointcuts, and advice. Andrew introduces a separate pointcut language based on logic meta programming (LMP) and does weaving via code transformations at the meta-level.

9 Summary and Final Remarks

AspectS is an approach to general-purpose AOP in the Squeak/Smalltalk environment with the intent to allow for experimental aspect-oriented system development. It represents an effort to help understand issues that come along with aspects in dynamic environments.

AspectS mainly draws on the results of AspectJ and MethodWrappers. It benefits greatly from the simple, elegant, and open architecture of Smalltalk.

AspectS allows for coordinated meta-level programming, addressing the tangled code phenomenon by providing aspect related modules. AspectS is realized using plain Smalltalk only, without extending neither the Smalltalk language nor its virtual machine. With that, AspectS utilizes the expressiveness of Smalltalk itself as a pointcut language. AspectS introduces on-demand weaving as a technique to perform system transformations back and forth whenever required.

Tool support for AspectS is available through the extension of Smalltalk code browsers to localize and examine all methods that are potentially affected by the application of an aspect or one of its advice, and to easily locate those parts of the system that have been affected after the application of aspects.

Acknowledgements

Thanks are due to Cristina Lopes for her great support, to David Simmons and Eliot Miranda for their insights and valuable remarks, to John Brant for his MethodWrappers package AspectS makes heavily use of, and to Dan Ingalls for solving a puzzle that allowed the port of MethodWrappers to Squeak. Thanks also to Matthias Wagner, Kris Gybels, and the anonymous reviewers for their comments on the paper.

References

1. AspectJ team: *AspectJ homepage* (http://aspectj.org)
2. Brant, J.; Foote, B.; Johnson, R.; Roberts, D.: *Wrappers to the Rescue.* In: ECOOP'98 Proceedings, 1998, pp. 396-417
3. Böllert, K.: *AOP/ST homepage* (http://www.theoinf.tu-ilmenau.de/~kaib/aop/)
4. Czarnecki, K.: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* Dissertation, TU Ilmenau, 1998 (http://www.prakinf.tu-ilmenau.de/~czarn/diss/)
5. de Alwis, B.: *Apostle homepage* (http://www.cs.ubc.ca/labs/spl/projects/apostle/)
6. Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983
7. Gybels, K.: *Andrew homepage* (http://prog.vub.ac.be/~kgybels/andrew/)
8. Hirschfeld, R.: *AspectS homepage* (http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/)
9. Ingalls, D. H. H.: *Design Principles Behind Smalltalk.* In: BYTE Magazine, August 1981
10. Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.: *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, 1993
11. JMangler team*: JMangler homepage* (http://javalab.cs.uni-bonn.de/research/jmangler)
12. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, Ch.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming.* In: ECOOP' 97 Proceedings, 1997, pp. 220-242
13. Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ.* In: ECOOP' 01 Proceedings, 2001, pp. 327-355
14. Lopes, C. V.: *D: A Language Framework for Distributed Programming.* Dissertation. College of Computer Science, Northeastern University, Boston, 1997
15. Mendhekar, A.; Kiczales, G.; Lamping, J.: *RG: A Case-Study for Aspect-Oriented Programming.* Xerox PARC. Technical Report SPL97-009 P9710044. February 1997
16. Brant, J,: *MethodWrappers homepage* (http://st-www.cs.uiuc.edu/~brant/Applications/MethodWrappers.html)
17. *Squeak homepage* (http://www.squeak.org)
18. *VisualWorks homepage* (http://www.parcplace.com, http://www.cincom.com)