

# AspectS – AOP with Squeak

Robert Hirschfeld  
hirschfeld@acm.org  
Cupertino, August 2001

## Abstract

AspectS is an approach to general-purpose aspect-oriented programming in the Squeak/Smalltalk environment. Based on the language model of AspectJ, it extends the Smalltalk MOP to accommodate the aspect modularity mechanism. In contrast to AspectJ, weaving in AspectS happens dynamically at runtime.

## 1 Introduction

Aspect-Oriented Programming (AOP) 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. As of today there are several approaches that support aspect-oriented concepts, ranging from general-purpose aspect languages like AspectJ [AJ01, Kicz+01] to domain-specific aspect languages such as RG or D [MKL97, Lope97].

AspectS<sup>1</sup> extends the Squeak/Smalltalk<sup>2</sup> environment to allow for experimental aspect-oriented system development [Hirs01]. 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 [BFJR99, MW01]. It benefits greatly from the simple, elegant, and open architecture of Squeak itself as well [GoRo83, Inga81, Sque01].

AspectS is based on the language model of AspectJ and represents an effort to help understand issues that come along with aspects in dynamic environments like Smalltalk. 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.

## 2 Aspects

Aspects are units of modularity that represent implementations of crosscutting concerns. Aspects are composed of several advice that associate code fragments (parts of a computational unit) with join points. Collection of related join points, to be addressed by advice, are called a pointcuts (Figure 1). Join points are targets for the weaving process.



Figure 1

In AspectS, aspects are implemented via regular classes, so their instances act as regular objects as well.

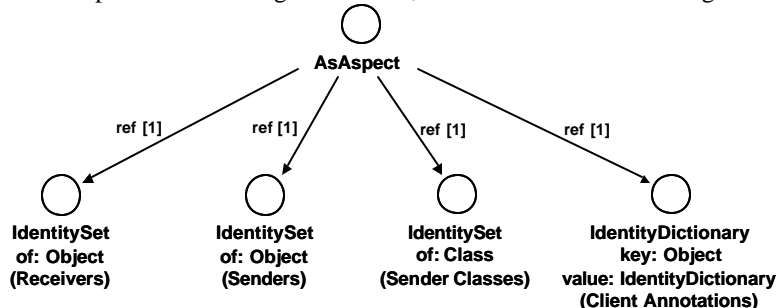


Figure 2

<sup>1</sup> The version of AspectS discussed in the text is 0.1.2 (2001-07-20, [Hirs01]).

<sup>2</sup> AspectS is implemented in Squeak. Squeak is an open, highly portable Smalltalk-80 implementation whose virtual machine is written entirely in Smalltalk. The terms Squeak and Smalltalk are used interchangeably in this text.

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 code at run-time to determine if receiver-instances-aware, sender-instance-aware 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 denote targets for the weaving process to apply computational changes to the underlying base system stated in advice.

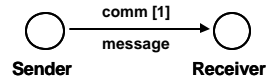


Figure 3

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 join point. A join point is described by a target class and a target selector (Figure 4).

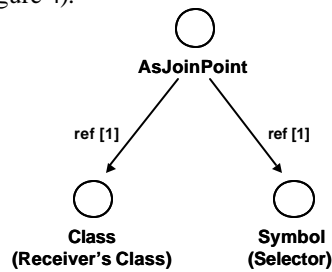


Figure 4

Pointcuts and their join points can be enumerated statically. Also, due to the very open and reflective nature of the Smalltalk environment, they can be collected dynamically as well by querying the system.

### 4 Advice

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

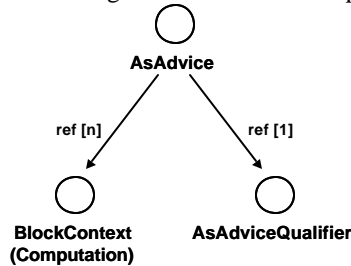


Figure 5

AspectS uses blocks to represent code fragments (Figure 5). Note that blocks in Squeak (instances of BlockContext) do not provide full closure semantics, which affects their application and makes certain assumptions about their use necessary.

Furthermore, advice are to be qualified to state if the woven code will be receiver-instances-aware, sender-instance-aware or sender-class-specific, combined with additional cflow 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)

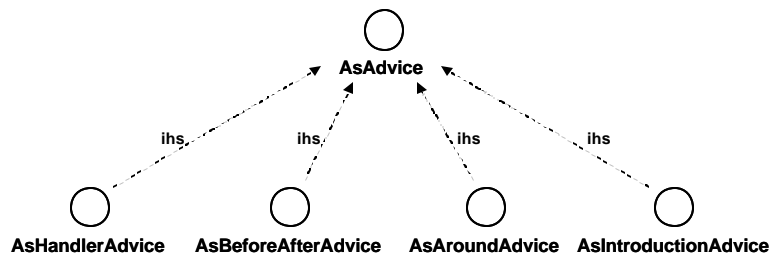


Figure 6

It is possible to *introduce* new behavior to the target clients as well (AsIntroductionAdvice).

**Handler Advice:** 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 Advice:** 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 Advice:** An around advice (AsAroundAdvice) may be put in front of a specific method to allow the conditional activation of that method with respect to actual execution context. The additional code, represented as a block context, has access to all method parameters.

**Introduction Advice:** With an introduction advice (AsAroundAdvice) 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.

## 4.2 Advice Qualifier

An advice qualifier (AsAdviceQualifier) is used to control the selection of an appropriate advice by offering a major and a minor attribute. The major attribute states sender/receiver aware activation, whereas the minor attribute adds cflow semantics. Advice qualifier attributes can be compared to AspectJ's concept of pointcut designators.

### Receiver/Sender-aware Activation

Each advice has to be qualified to be either receiver-general, receiver-specific, sender-general, or sender-specific. This is done via the major attribute of the advice qualifier.

**Receiver-general Advice:** With a receiver-general advice, all receivers of the message that are of a certain class are going to be affected. The advice is receiver-class-aware.

**Receiver-specific Advice:** With a receiver-specific advice, only specific receivers of the message that are 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-general Advice:** With a sender-general advice, receivers of the message that are of a certain class are going to be affected if the sender is of a certain class. Sender classes can be added to or removed from the advice's aspect. The advice is sender-class-aware.

**Sender-specific Advice:** With a sender-specific advice, receivers of the message that are 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

An advice may be further qualified by the minor attribute of the advice qualifier to add cflow activation semantics. Cflow is currently limited to class- and instance-based method- and object-recursion, but will be extended to the more generic model supported by AspectJ.

**Class-specific – First:** With a class-specific-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-specific – All-But-First:** With a class-specific 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-specific – First:** With a instance-specific-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-specific – All-But-First:** With a instance-specific 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.)

## 5 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. 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.

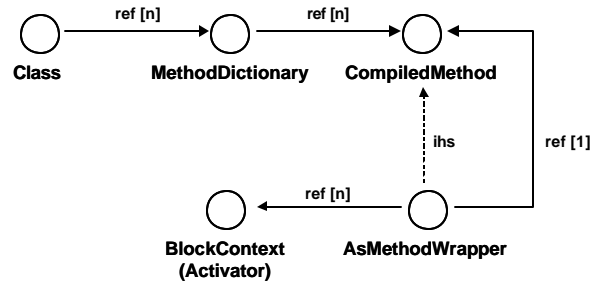


Figure 7

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).

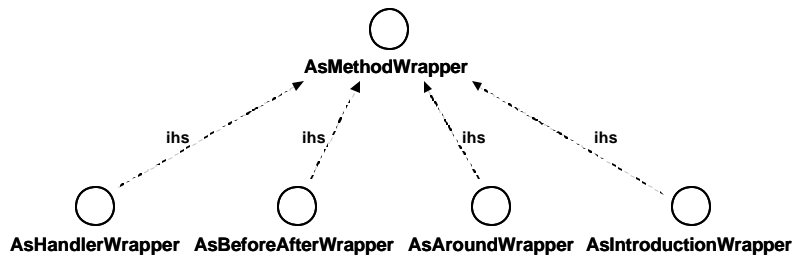


Figure 8

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).

### 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).

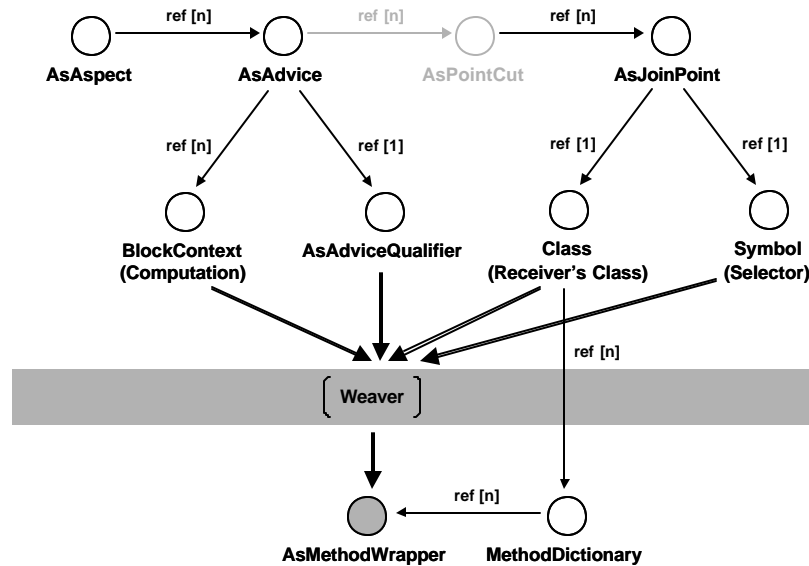


Figure 9

The weaving process happens every time an aspect instance is installed. To reverse the effects of an aspect to the system, the aspect has to be uninstalled. This process is also referred to as unweaving. Weaving 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.

## 6 Example

The following example illustrates the basic mechanisms of AspectS. 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#'` 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.

In `adviceMouseEnter` (see the following code snippet), 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.

To activate the `AsMorphicMousingAspect`, one creates an aspect instance sends it an `install` message. The send of an `uninstall` message deactivates the aspect.

```

adviceMouseEnter
  ^ AsBeforeAfterAdvice new
    qualifier: (AsAdviceQualifier major: #receiverGeneral minor: nil);
    joinPoints: (
      (Smalltalk allClassesImplementing: #mouseEnter:)
        select: [:each | each includesBehavior: Morph]
        thenCollect: [:each |
          AsJoinPoint new
            targetClass: each;
            targetSelector: #mouseEnter:]);
    beforeBlock: [:receiver :arguments :aspect :client |
      self
        showHeader: '>>> MouseENTER >>>'
        receiver: receiver
        event: arguments first]

```

## 7 Final Remarks

Tool support for AspectS is at this time is mainly centered around the extension of Smalltalk code browsers to browse all the 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.

Other efforts to provide AOP support for Smalltalk environments are AOP/ST for VisualWorks by Kai Böllert and Apostle for VisualAge for Smalltalk by Brian de Alwis [Böll01, deAl01]. 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.

## Acknowledgements

Thanks are due to Cristina Lopes, David Simmons, and Eliot Miranda for their insights and valuable comments, 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.

## 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
- [Böll01] Böllert, Kai: AOP/ST homepage (<http://www.theoinf.tu-ilmeneau.de/~kaib/aop/>)
- [deAl01] de Alwis, Brian: Apostle homepage (<http://www.cs.ubc.ca/labs/spl/projects/apostle/>)
- [GoRo83] Goldberg, Adele; Robson, David: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983
- [Hirs01] Hirschfeld, Robert: AspectS homepage (<http://www.prakinf.tu-ilmeneau.de/~hirsch/Projects/Squeak/AspectS/>)
- [Inga81] Ingalls, Daniel H. H.: *Design Principles Behind Smalltalk*. In: BYTE Magazine, August 1981
- [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
- [Lope97] Lopes, Cristina Videira: *D: A Language Framework for Distributed Programming*. Dissertation. College of Computer Science, Northeastern University, Boston, 1997
- [MKL97] Mendhekar, Anurag; Kiczales, Gregor; Lamping, John: *RG: A Case-Study for Aspect-Oriented Programming*. Xerox Palo Alto Research Center. Technical Report SPL97-009 P9710044. February 1997
- [MW01] MethodWrappers homepage (<http://st-www.cs.uiuc.edu/~brant/Applications/MethodWrappers.html>)
- [Sque01] Squeak homepage (<http://squeak.org>)