An Introduction to Context-Oriented Programming with ContextS

Robert Hirschfeld[1], Pascal Costanza[2], and Michael Haupt[1]

[1] Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany
{robert.hirschfeld,michael.haupt}@hpi.uni-potsdam.de
[2] Programming Technology Lab, Vrije Universiteit Brussel, B-1050 Brussels, Belgium
pascal.costanza@vub.ac.be

Abstract. Context-oriented Programming, or COP, provides programmers with dedicated abstractions and mechanisms to concisely represent behavioral variations that depend on execution context. By treating context explicitly, and by directly supporting dynamic composition, COP allows programmers to better express software entities that adapt their behavior late-bound at run-time. Our paper illustrates COP constructs, their application, and their implementation by developing a sample scenario, using ContextS in the Squeak/Smalltalk programming environment.

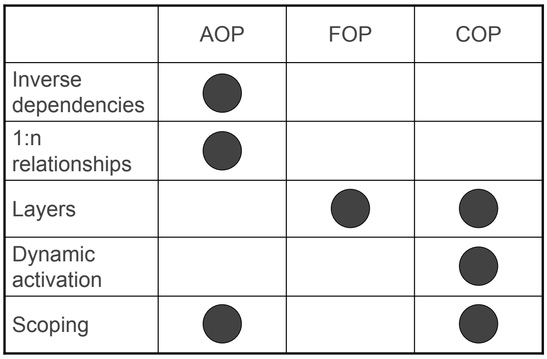
1 Introduction

Every intrinsically complex application exhibits behavior that depends on its context of use. Here, the meaning of context is broad and can range from obvious concepts such as location, time of day, or temperature over more technical properties like connectivity, bandwidth, battery level, or energy consumption to a user's subscriptions, preferences, or personalization in general.

Besides these examples of context that are often associated with the domain of ambient computing, the computational context of the program itself, for example its control flow or the sets or versions of libraries used, can be an important source of information for affecting the behavior of parts of the system.

Even though context is a central notion in a wide range of application domains, there is no direct support of context-dependent behavior from traditional programming languages and environments. Here, the expression of variations requires developers to repeatedly state conditional dependencies, resulting in scattered and tangled code.

This phenomenon, also known as crosscutting concerns, and some of the associated problems were documented by the aspect-oriented programming (AOP [16]) and the feature-oriented programming (FOP [2]) communities. The focus of AOP is mainly on the establishments of inverse one-to-many relationships [17] to achieve their vision of quantification and obliviousness [10]. FOP's main concern is the compile-time selection and combination of variations, and the necessary algebraic means to reason about such layer compositions [3].



	AOP	FOP	COP
Inverse dependencies	●		
1:n relationships	●		
Layers		●	●
Dynamic activation			●
Scoping	●		●

Fig. 1. Properties of AOP, FOP, and COP

Context-oriented programming (COP [6,14]) addresses the problem of dynamically composing context-dependent concerns, which are potentially crosscutting. COP takes the notion of FOP layers and provides means for their selection and composition at run-time. While FOP mechanisms are applied at compile-time—with the effect that, during program execution, layers as a distinct entity are no longer available—, COP preserves layers, adds the notion of dynamic layer activation and deactivation, and provides dynamic scoping to delimit the visibility of their composition as needed (Figure 1). With the dynamic scoping mechanisms offered by COP implementations, layered code can be associated with the units it belongs to and can be composed into or removed from the system depending on its context of use.

There are several COP extensions to popular programming languages such as ContextL for Lisp [6], ContextS for Squeak/Smalltalk [14], ContextR for Ruby, ContextPy for Python, and ContextJ* for Java [14]. Here, we will focus on ContextS. Our paper is meant to be used mainly as a tutorial, describing ContextS in how it can be applied to the implementation of context-dependent behavioral variations.

The remainder of our paper is organized as follows: We give an overview of COP in Section 2. In Section 3 we introduce some of the COP extensions provided with ContextS which are applied to an example presented in Section 4. After some recommendations for further reading in Section 5 we conclude our paper with Section 6.

2 Context-Oriented Programming

COP, as introduced in [6,14], facilitates the modularization of context-dependent behavioral variations. It provides dedicated programming abstractions and mechanisms to better express software entities that need to change their behavior depending on their context of use.

Based on the implementation of several application scenarios and the development of language extensions necessary for them, we have identified behavioral variations, layers, dynamic activation, contextual information, and proper scoping mechanisms as essential properties of COP support:

Behavioral variations. There is a means to specify behavioral variations, typically ranging from new to modified or even removed behavior. Here, partial definitions of modules of the underlying programming model such as procedures, methods, or classes are prime candidates for being expressed as behavioral variations.

Layers. There needs to be a means to group related behavioral variations into layers. As first-class entities, layers can be explicitly referred to at run-time.

Activation/deactivation. Individual layers or combinations of them can be dynamically activated and deactivated, giving explicit control to programmers over their composition – including the point in time of their activation/deactivation as well as the desired sequence of their application.

Context. COP adopts a very broad definition of context: Context is everything that is computationally accessible. With that, we do not limit context to a particular concept, but encourage a wide spectrum of context representations most suitable for a specific application or system.

Scope. The scope of a layer activation or deactivation can be controlled explicitly so that simultaneous compositions affect each other only to the degree required by the program.

In the following, we will use message dispatch to show how COP is a continuation of previous work. While we do not require message dispatch as a base for any COP implementation, we do believe that this illustration will help to better understand how COP builds on procedural, object-oriented, and subjective programming (Figure 2).

1D dispatch. *Procedural programming* offers only one dimension to associate a unit of behavior with its activation [18]. Names used in procedure calls are directly mapped to procedure implementations (<*m*>, Figure 2a).

2D dispatch. *Object-oriented programming* already uses two dimensions to associate a unit of behavior with its activation [18]. Names used at the activation site are mapped to a method implementation with the same name and the receiver it is defined in (<*m, R*>, Figure 2b).

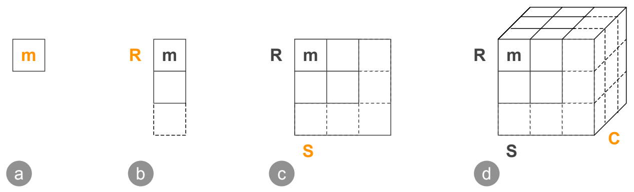


Fig. 2. Multi-dimensional Message Dispatch

3D dispatch. *Subjective programming* as introduced in [18] goes one step further than object-oriented programming in that it adds a third dimension to message dispatch. Here, method implementations are selected not only by their name and the receiver they are defined in, but also the sender the message send originated from (<*m, R, S*>, Figure 2c).

4D dispatch. *COP* considers yet another dimension by dispatching not only on the name of a behavioral unit, the receiver it is defined in, and the sender the message originated from, but also on the context of this particular message send (<*m, R, S, C*>, Figure 2d).

3 ContextS

ContextS is our COP extension to Squeak/Smalltalk to explore COP in late-bound class-based programming environments [12,15]. In Squeak/Smalltalk there are only objects, and messages exchanged between them. Since everything else is built on top of these concepts, and due to late-binding being used extensively throughout the system, the realization of ContextS was simple and straightforward. Only small changes to the language kernel needed to be made to achieve useful results. In this section we give a brief introduction to the small set of constructs provided with ContextS, leaving the illustration of their application to Section 4. We try to refrain from discussing implementation details, but will mention some alternatives we are currently investigating.[1]

3.1 Implementation-Side Constructs

There are two main concepts to be used at the implementation side of concerns implemented in ContextS: Layers and advice-based partial method definitions.

Layers are simply represented as subclasses of `CsLayer`. In current versions of ContextS, layers are containers for partial method definitions. In future versions, we will move such method definitions away from the layers, into the classes they belong to. For a detailed discussion on why that should be done, please refer to our reading list, presented in Section 5.

```
CsLayer subclass: #MyLayer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'My Category'
```

Partial *method* definitions, as shown here, still use an advice-based style as introduced in PILOT [19] and popularized by CLOS [4] and AspectJ-style language extensions [16]. The style presented here is inherited from AspectS [13], to take advantage of a set of metaobjects called *method wrappers* AspectS was built on [5].

[1] We use ContextS version 0.0.10 throughout the paper, available from http://www.swa.hpi.uni-potsdam.de/cop/.

```
MyLayer>>adviceCopLeafEvaluate
  ^ CsAroundVariation
    target: [MyTargetClass -> #myTargetSelector]
    aroundBlock: [:receiver :arguments :layer :client :clientMethod |
      "my layer-specific code"
      ...]
```

Here we can see that a partial method definition (`adviceCopLeafEvaluate`) belongs to a particular layer (`MyLayer`). The name of each such method definition needs to start with `advice` and has to have no arguments. These properties are exploited by the underlying framework to collect and compose all partial definitions associated with a layer. With `CsAroundVariation` we state that we will apply an *around* advice with class `MyTargetClass` and method `myTargetSelector` as its target. Layer-specific code provided by this partial method definition is stated in the *around* block and has full access to its environment, including the sender and the receiver of the message, its arguments, as well as the defining layer.

3.2 Activation-Side Constructs

At the client side of a concern implemented using ContextS, `useAsLayersFor:` is about the only construct ever used.

```
receiver useAsLayersFor: argument
```

`useAsLayersFor:` is a regular message that can be sent to collections or arrays that contain instances of `CsLayer`, or to code blocks that eventually return such collections or arrays.

All layers (instances of `CsLayer`) enumerated or computed by the receiver object are composed into the Squeak image in the order of their appearance in the list. This composition is only effective in the current process (Squeak's version of a thread) and for the dynamic extent of the execution of the block (an instance of `BlockContext` which is provided as the second argument).

4 Pretty-Printing as an Example

We use the task of pretty-printing an expression tree in infix, prefix, and postfix notation as well as an evaluation of the same as an example to show the differences between a regular object-oriented solution, an approach using the Visitor design pattern [11], and our context-oriented version.

The basic implementation of the nodes and leaves used to construct our expression trees is shown in Figure 3. A `CopNode` (left-hand side of Figure 3) has three instance variables for the first operand, the second operand, and the operation to combine the two. Each `CopLeaf` (right-hand side of Figure 3) has only one instance variable providing its value. Both classes provide accessor methods for their instance variables.

An expression tree can be assembled by the creation of individual instances of `CopNode` and `CopLeaf` and their combination. The example tree used throughout

```
Object subclass: #CopNode
  instanceVariableNames: 'first op second'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ContextS-Demo Visitor'

CopNode class>>first: aFirstCopNodeOrCopLeaf
    op: aSymbol second: aSecondCopNodeOrCopLeaf
  ^ self new
    first: aFirstCopNodeOrCopLeaf;
    op: aSymbol;
    second: aSecondCopNodeOrCopLeaf

CopNode>>first
  "^ <CopNode | CopLeaf>"
  ^ first

CopNode>>first: anCopNodeOrCopLeaf
  first := anCopNodeOrCopLeaf.

CopNode>>op
  "^ <Symbol>"
  ^ op

CopNode>>op: aSymbol
  op := aSymbol.

CopNode>>second
  "^ <CopNode | CopLeaf>"
  ^ second

CopNode>>second: anCopNodeOrCopLeaf
  second := anCopNodeOrCopLeaf.
```

```
Object subclass: #CopLeaf
  instanceVariableNames: 'value'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ContextS-Demo Visitor'

CopLeaf class>>value: anInteger
  ^ self new value: anInteger

CopLeaf>>value
  "^ <Integer>"
  ^ value

CopLeaf>>value: anInteger
  value := anInteger.
```

Fig. 3. Basic Node and Leaf Implementation

```
tree := CopNode
  first: (CopNode
    first: (CopNode
      first: (CopLeaf value: 1)
      op: #+
      second: (CopLeaf value: 2))
    op: #*
    second: (CopNode
      first: (CopLeaf value: 3)
      op: #-
      second: (CopLeaf value: 4)))
  op: #/
  second: (CopLeaf value: 5).
```

Fig. 4. Construction of an Expression

our paper is built in Figure 4. Figure 5 provides a graphical representation of the tree created in Figure 4.

4.1 Regular Objects

A simple and straightforward object-oriented implementation of pretty-printing is listed in Figure 6. The desired behavior is implemented *in-place* in both classes as methods `evaluate`, `printInfix`, `printPostfix`, and `printPrefix` respectively.

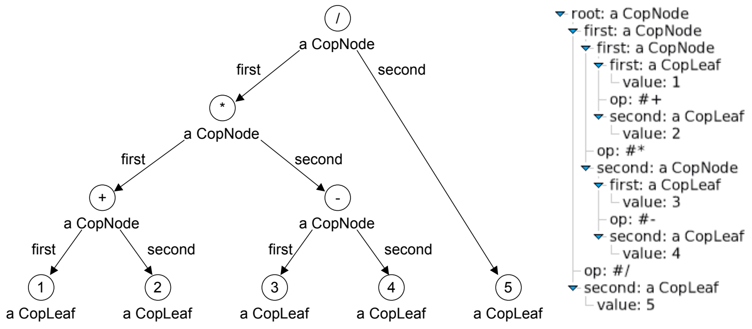


Fig. 5. Expression Tree

```
CopNode>>evaluate                                              CopLeaf>>evaluate
  ^ (self first evaluate)                                        ^ self value
    perform: self op with: (self second evaluate)

CopNode>>printInfix                                            CopLeaf>>printInfix
  ^ '(', self first printInfix,                                  ^ self value asString
    self op, self second printInfix, ')'

CopNode>>printPostfix                                          CopLeaf>>printPostfix
  ^ '(', self first printPostfix,                                ^ self value asString
    self second printPostfix, self op, ')'

CopNode>>printPrefix                                           CopLeaf>>printPrefix
  ^ '(', self op,                                                ^ self value asString
    self first printPrefix, self second printPrefix, ')'
```

Fig. 6. In-place Traversals

```
Transcript cr; show: tree printInfix.       ==> (((1+2)*(3-4))/5)
Transcript cr; show: tree printPrefix.      ==> (/(*(+12)(-34))5)
Transcript cr; show: tree printPostfix.     ==> (((12+)(34-)*)5/)
Transcript cr; show: tree evaluate.         ==> (-3/5)
```

Fig. 7. Use of In-place Traversals

Because these methods need to coexist side-by-side at the same time, they are named differently. And because of that, client side code needs to explicitly decide which one to use.

An application of our system so far is copied down in Figure 7, with the code executed on its left-, and the resulting print-outs on its right-hand side. In Squeak, objects are printed to the system console called `Transcript` by sending it the message `show:` with the object as argument.

4.2 Visitors

Our solution to the implementations of pretty-printing as presented previously is used as a motivation for the Visitor design pattern [11]. From its intent, we take that a visitor represents "an operation to be performed on the elements of an object structure" where the Visitor makes it easy to add new operations to

```
CopNode>>accept: aCopVisitor        CopLeaf>>accept: aCopVisitor
  ^ aCopVisitor visitNode: self       ^ aCopVisitor visitLeaf: self
```

Fig. 8. Visitor-ready Base Objects

the entire structure. This is because all new operations can be defined outside this object structure it operates on, and so without the need to change it.

However, the resulting client code of Visitor-based systems is very hard to understand, with the manual simulation of double dispatch being one of the main reasons for that. Figure 8 lists the basic framework used for that in `CopNode` and `CopLeaf`, adding implementations of `accept:` to both classes that then call back to the argument, providing itself and the proper type information needed for further processing.

The actual implementation of two of the four visitors, `CopPrintPrefixVisitor` and `CopEvaluateVisitor`, can be seen in Figure 9. Here, our `visitLeaf:` and `visitNode:` methods control both local computation and follow-up traversals.

```
CopVisitor subclass: #CopPrintPrefixVisitor     CopVisitor subclass: #CopEvaluateVisitor
  instanceVariableNames: ''                        instanceVariableNames: ''
  classVariableNames: ''                           classVariableNames: ''
  poolDictionaries: ''                             poolDictionaries: ''
  category: 'ContextS-Demo Visitor'               category: 'ContextS-Demo Visitor'

CopPrintPrefixVisitor>>visitLeaf: aCopLeaf      CopEvaluateVisitor>>visitLeaf: aCopLeaf
  ^ aCopLeaf value asString                       ^ aCopLeaf value

CopPrintPrefixVisitor>>visitNode: aCopNode      CopEvaluateVisitor>>visitNode: aCopNode
  ^ '(',                                          ^ (aCopNode first accept: self)
    aCopNode op,                                    perform: aCopNode op
    (aCopNode first accept: self),                  with: (aCopNode second accept: self)
    (aCopNode second accept: self),
    ')'
```

Fig. 9. Visitor Examples

```
Transcript cr; show: (tree accept: CopPrintInfixVisitor new).       ==> (((1+2)*(3-4))/5)
Transcript cr; show: (tree accept: CopPrintPrefixVisitor new).      ==> (/(*(+12)(-34))5)
Transcript cr; show: (tree accept: CopPrintPostfixVisitor new).     ==> (((12+)(34-)*)5/)
Transcript cr; show: (tree accept: CopEvaluateVisitor new).         ==> (-3/5)
```

Fig. 10. Use of Visitors

An application of our Visitor-based system so far is transcribed in Figure 10, again with the code executed on its left, and the resulting print-outs on its right.

4.3 Layers

Now we are going to implement our expression traversal example using ContextS by applying the constructs introduced in Section 3. Here, we present two of our four layers, `CopPrintInfixLayer` and `CopEvaluateLayer`, in Figure 11. Each layer is a subclass of `CsLayer`, provides two partial method definitions with class `CopLeaf`, class `CopNode`, and their methods `printOn:` as targets. As

```
CsLayer subclass: #CopPrintPrefixLayer
  instanceVariableNames: ’’
  classVariableNames: ’’
  poolDictionaries: ’’
  category: ’ContextS-Demo Visitor’

CopPrintPrefixLayer>>adviceCopLeafPrintOn
  ˆ CsAroundVariation
    target: [CopLeaf -> #printOn:]
    aroundBlock: [:receiver :arguments
        :composition :client :clientMethod |
      receiver value printOn: arguments first]

CopPrintPrefixLayer>>adviceCopNodePrintOn
  | stream |
  ˆ CsAroundVariation
    target: [CopNode -> #printOn:]
    aroundBlock: [:receiver :arguments
        :layer :client :clientMethod |
      stream := arguments first.
      stream nextPut: $(.
      stream nextPutAll: receiver op.
      receiver first printOn: stream.
      receiver second printOn: stream.
      stream nextPut: $)]
```

```
CsLayer subclass: #CopEvaluateLayer
        instanceVariableNames: ’’
        classVariableNames: ’’
        poolDictionaries: ’’
        category: ’ContextS-Demo Visitor’

      CopEvaluateLayer>>adviceCopLeafEvaluate
        ˆ CsAroundVariation
          target: [CopLeaf -> #evaluate]
          aroundBlock: [:receiver :arguments
              :composition :client :clientMethod |
            receiver value]

      CopEvaluateLayer>>adviceCopNodeEvaluate
        ˆ CsAroundVariation
          target: [CopNode -> #evaluate]
          aroundBlock: [:receiver :arguments
              :layer :client :clientMethod |
            receiver first evaluate
              perform: receiver op
              with: receiver second evaluate]
```

Fig. 11. Layered Traversal Code

```
[ { CopPrintInfixLayer new } ] useAsLayersFor: [
  Transcript cr; show: tree].                          ==> (((1+2)*(3-4))/5)

[ { CopPrintPrefixLayer new } ] useAsLayersFor: [
  Transcript cr; show: tree].                          ==> (/(*(+12)(-34))5)

[ { CopPrintPostfixLayer new } ] useAsLayersFor: [
  Transcript cr; show: tree].                          ==> (((12+)(34-)*)5/)

[ { CopEvaluateLayer new } ] useAsLayersFor: [
  Transcript cr; show: tree evaluate].                 ==> (-3/5)
```

Fig. 12. Use of Layered Traversal Code

already stated previously, objects are printed to the `Transcript` by sending it the message `show:` with the object as argument. `show:` itself sends `printOn:` to the object, with a stream as its argument. This is the method we would override in a subclass to change the behavior of `show:`, and this is also the method we adapt using COP-style refinements.

In this example, our context-dependent behavior simply overrides the original behavior present before its layer activation by using an around construct without a proceed. More method combinations including around with proceed, before, or after semantics can be achieved as well, but are beyond the scope of this tutorial.

In an upcoming version of ContextS, the need for AOP-style inverse relationships will be reduced, since our traversal code belongs to the objects and should be defined in the scope of their classes so that programmers reading their code are aware of its impact (Figure 13).

The activation-side of our context-dependent behavior looks as simple and straightforward as promised (Figure 12): The programmer of the client code states or computes the desired layer combination (only one layer in our example), and uses it via the `useAsLayersFor:` message. This causes the layer activation to be composed into the system and visible in the dynamic extent of the execution of the provided block argument.

Please note that the provided block arguments are the same in all four cases. The method `show:` is used all the time to print out our expression tree. Only the layer used is different from case to case. It is also important to point out that in the code block provided to `useAsLayersFor:` the message `show:` is sent to `Transcript` whereas our layers adapt `printOn:` of `CopLeaf` and `CopNode` respectively.

5 Further Reading

Related work of COP including AOP, FOP, or delegation have been presented and discussed in other publications In the following we list some of them for further reading:

Language Constructs for Context-oriented Programming – An Overview of ContextL [6]. That paper presents ContextL, our first COP extension. It supplements the Common Lisp Object System (CLOS) with layers, layered classes, layered and special slots, layered accessors, and layered functions. In ContextL, layered classes, slots and functions can be accumulated in layers. ContextL's layers or their combinations are dynamically scoped, allowing us to associate partial behavioral variations with the classes they belong to, while, at the same time, changing their specific behavior depending on the context of their use.

Efficient Layer Activation for Switching Context-dependent Behavior [8]. In that paper, we illustrate how ContextL constructs can be implemented efficiently. As an interesting result, ContextL programs using repeated layer activations and deactivations are about as efficient as without, underlining the fact that apparently other things are more important regarding performance and its optimization. We also show an elegant and efficient implementation of the prominent AOSD figure editor example, even without the need to resort to cflow-style constructs in the first place.

Reflective Layer Activation in ContextL [7]. That paper describes a reflective approach to the expression of complex, application-specifc layer dependencies, without compromising efficiency.

Context-oriented Programming [14]. In that contribution, we summarize our previous work and present COP as a novel approach to the modularization of context-dependent behavior. We show that, by treating context explicitly and by providing dedicated abstractions and mechanisms to represent context-dependent variations, programs can be expressed more concisely than without. Several examples are provided to illustrate COP's advantages over more traditional approaches.

```
CopNode>>printOn: aStream          CopNode>>printOn: aStream
  <<layer: PrintPrefix>>             <<layer: Evaluate>>
  ^ ’(’,                             ^ self first evaluate
    self op,                           perform: self op
    self first printPrefix,            with: (self second evaluate)
    self second printPrefix,
    ’)’

CopLeaf>>printOn: aStream          CopLeaf>>printOn: aStream
  <<layer: PrintPrefix>>             <<layer: Evaluate>>
  ^ self value asString              ^ self value
```

Fig. 13. Another Representation of Layered Traversal Code

6 Summary and Outlook

In our tutorial-style introduction to ContextS we have presented both activation-side and implementation-side constructs of our COP extension to Squeak/ Smalltalk. We provided sample implementations illustrating three different approaches to printing out an expression tree: plain and straightforward object-oriented programming, a Visitor-based application, as well as a COP-based solution using ContextS.

Our intent was not to discuss advantages of COP-style development but rather to present some of the mechanics involved in working with ContextS. One of the issues we are working on right now is to allow for specifying context-specific code outside of layers and inside the classes it belongs to. Figure 13 presents one of our approaches to this issue. Here we can see different versions of the `printOn:` method, associated with one and the same class at the same time. The difference is the layer expressed via `<<layer: ...>>` denoting the layer the particular method belongs to.

While ContextS provides means to express heterogeneous crosscutting behavioral variations, we look into its extension to concisely implement homogeneous crosscutting concerns as well [1].

We are currently working on medium- to large-sized examples and case studies to illustrate to what extent COP helps to better express software entities that need to adapt their behavior to their context of use, and to further refine our language extensions.

Acknowledgements

We thank Marcus Denker, Tudor Girba, Robert Krahn, Dave Thomas, and Jan Wloka for their fruitful discussions and valuable contributions, and Ralf Lämmel for his patience.

References

1. Apel, S.: The Role of Features and Aspects in Software Development. PhD thesis, Otto-von-Guericke University Magdeburg (March 2007)
2. Batory, D.: Feature-oriented programming and the ahead tool suite. In: Proceedings of the International Conference on Software Engineering 2004 (2004)

3. Batory, D., Rauschmeyer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering (June 2004)
4. Daniel, G., Bobrow, L.G., De Michiel, R.P., Gabriel, S.E.: Common lisp object system specification: 1. programmer interface concepts. Lisp and Symbolic Computation 1(3-4), 245–298 (1989)
5. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the rescue. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
6. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming — an overview of ContextL. In: Wuyts, R. (ed.) Proceedings of the 2005 Dynamic Languages Symposium, ACM Press, New York (2005)
7. Costanza, P., Hirschfeld, R.: Reflective layer activation in ContextL. In: Proceedings of the Programming for Separation of Concerns (PSC) of the ACM Symposium on Applied Computing (SAC). LNCS. Springer, Heidelberg (2007)
8. Costanza, P., Hirschfeld, R., De Meuter, W.: Efficient layer activation for switching context-dependent behavior. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS, vol. 4228. Springer, Heidelberg (2006)
9. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
10. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman et al., [9], pp. 21–35.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1995)
12. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston (1983)
13. Hirschfeld, R.: AspectS – aspect-oriented programming with Squeak. In: Akşit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 216–232. Springer, Heidelberg (2003)
14. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology (JOT) 7(3), 125–151 (2008)
15. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of squeak, a practical smalltalk written in itself. In: OOPSLA 1997: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 318–326. ACM Press, New York (1997)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
17. Nordberg III, M.E.: Aspect-oriented dependency management. In: Filman et al., [9], pp. 557–584
18. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS special issue on Subjectivity in Object-Oriented Systems 2(3), 161–178 (1996)
19. Teitelman, W.: Pilot: A step towards man-computer symbiosis. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1966)