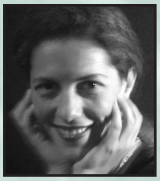
ISR Institute for Software Research
University of California, Irvine

Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)



Cristina Videira Lopes
University of California, Irvine
lopes@ics.uci.edu

December 2002

ISR Technical Report # UCI-ISR-02-5

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)

Cristina Videira Lopes
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
lopes@ics.uci.edu

ISR Technical Report # UCI-ISR-02-5

December 2002

Abstract: The term "Aspect-Oriented Programming" (AOP) came into existence sometime between November of 1995 and May of 1996, at the Xerox Palo Alto Research Center (PARC). AOP was based on an extensive body of prior work, but somehow the existing terminology wasn't appropriate for describing what we were doing. The new programming technology we were beginning to devise was going to change the world! In this article I will give my own account of how AOP – the ideas, the technologies and the name – came to be. But History is just marginally interesting if one doesn't make the effort to learn from it and apply that knowledge in things that are still to come. AOP didn't quite "change the world" but, no doubt, it had an impact in research communities and in programming at- large. There are valuable lessons to be learned from the emergence of AOP, and an analysis of those is the ultimate goal of this article.

UCI-ISR-02-5

Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)

Cristina Videira Lopes
Information and Computer Science
University of California, Irvine
Irvine, CA 92697
lopes@ics.uci.edu

Abstract

The term "Aspect-Oriented Programming" (AOP) came into existence sometime between November of 1995 and May of 1996, at the Xerox Palo Alto Research Center (PARC). AOP was based on an extensive body of prior work, but somehow the existing terminology wasn't appropriate for describing what we were doing. The new programming technology we were beginning to devise was going to change the world!

In this article I will give my own account of how AOP – the ideas, the technologies and the name – came to be. But History is just marginally interesting if one doesn't make the effort to learn from it and apply that knowledge in things that are still to come. AOP didn't quite "change the world" but, no doubt, it had an impact in research communities and in programming at-large. There are valuable lessons to be learned from the emergence of AOP, and an analysis of those is the ultimate goal of this article.

1 A Matter of Style

Giving an historical perspective of a technology involves stating facts as much as it involves describing the technical context in which the technology emerged and understanding the dynamics of the group of people who created it. Having been part of that group, I am in a privileged position to tell the AOP story, or at least a rendition of that story. Like most historical perspectives, this one is based on facts, but the interpretations and comments are entirely my own. Also like most historical perspectives, this one is incomplete: it focuses on the period 1995-1998, the time when AOP and, later, AspectJ emerged. A lot had happened before and a lot has happened since then.

AOP, as such, started emerging in 1995 when I was already at PARC as a visiting student. In the years that followed, one of the types of questions I got asked more often was "what is AOP?" Is it a programming language? Macros in disguise? A design methodology? A clever pre-processor? Meta-programming? How is this different from X (replace X with your favorite programming trick or language feature)?… The other question I got asked frequently was "what are aspects?" Synchronization and tracing *feel* like aspects but what else? And what makes an aspect be an aspect, anyway? …

As I'm writing this article, seven years later, I have good answers for all of these questions. But that wasn't the case back in 1995-1998. In fact, many brilliant minds have blamed the AOP group for propagating subversive ideas without having clear definitions

Copyright © 2002 by the author. All rights reserved.

for what we were trying to do. They were right: our definitions were fuzzy and got clearer over time. Back then, we had two options: we could lock ourselves in the office for a few years, brainstorm and beat the thing to death until we figured it all out; or we could bring a semi-baked idea to the public and iterate with a larger community until the clear definitions would emerge. We chose the latter. The reasons for this choice are as much pragmatic as they are a matter personal style. The pragmatic reasons included the following. First, we all believed that the validation of the AOP thesis (i.e. that it led to better programs) could only be done outside the controlled environment of our offices. So there was no point in locking ourselves up to come up with a beautiful formal semantics, because that would miss the core of the thesis. Second, we also believed that what we were doing crosscut the boundaries of the traditional communities in software engineering and programming languages. We needed to get early input from different kinds of people, especially from "real programmers," our ultimate valuators.

Many researchers will probably resonate with this need to reach out in order to validate their work; others won't. Again, a matter of style. But among those that do, not many can do it successfully, even when their work is impressive. It takes financial support, a good team and a good team leader – these are all management issues that many researchers tend to overlook. The popularity of AOP and AspectJ is due, firstly, to Gregor Kiczales, not only for his technical leadership but also to his natural ability to secure resources and attract people.

2 Research Trends in the Early 90s

In writing this section, I have consulted a technical report written by my colleague Walter Hürsch and myself at the end of 1994 (Hürsch and Lopes 1995). That report was entitled "Separation of Concerns" and it was written when I was still at Northeastern University. In retrospect, it is evident that we missed a few important pieces of work and that we had a somewhat narrow vision of what it was that was being separated. But overall, that report did a good job in capturing a trend that was in the air, and that is the reason why I am evoking it here. People were talking about "separation of concerns;" our report captured the building blocks and the conceptual glue of what later became AOP and the AOP community. For reasons that fall out of the scope of this article, Walter and I stopped working on that report. I continued working on my thesis, which focused in concurrency and distribution Aspects.

What follows is a summary of that report. Parts in italics are verbatim text. In reading it, the reader should place him/herself in 1994.

2.1 Formulation of the Problem

The increasing complexity of today's software applications and the advent of novel and innovative technology make it necessary for programs to incorporate and deal with an ever greater variety of special computing concerns such as concurrency, distribution, real-time constraints, location control, persistence, and failure recovery. Underlying all of these special purpose concerns is the basic concern responsible for the fundamental computational algorithm and the basic functionality. Special purpose concerns exist to either fulfill special requirements of the application (real-time, persistence, distribution),

Copyright © 2002 by the author. All rights reserved.

or manage and optimize the basic computational algorithm (location control, concurrency).

Typical approaches to integrate an additional concern have been to extend a given programming language by providing a few new programming language constructs that address the concern. An example of such an extension is the Distributed Real-time Object Language DROL (Takashio and Tokoro 1992), an extension of C++ with the capability of describing distributed real-time systems.

Even though the concerns may be separated conceptually and incorporated correctly, commingling them in the code brings about a number of problems:

- *Programming intertwined code is hard and complex since all concerns have to be dealt with at the same time and at the same level. The extended programming language provides no adequate abstraction of concerns at the implementation level.*
- *Intertwined code is hard to understand because of the above lack of abstraction.*
- *Commingled code is hard to maintain and modify because the concerns are strongly coupled.*
- *Specific to object-oriented systems, the intertwined code gives rise to inheritance anomalies (Aksit et al. 1994, Matsuoka and Yonezawa 1993) due to the strong coupling of the different concerns. It becomes impossible to redefine a method implementation or the commingled special concern in a subclass without redefining both.*

Many researchers have recognized the above problems for single concerns in their specific area of expertise and have started to address them. Many devised techniques for separating individual concerns (Aksit et al. 1992, Aksit et al. 1994, Honda and Tokoro 1992, Okamura and Ishikawa 1994, Aksit, Wakita et al. 1994, Lopes and Lieberherr 1994, Lieberherr et al. 1994).

2.2 Analysis of the Problem and Specialized Solutions

For software concerns, we distinguished two different levels of separation:

Conceptual level. *At the conceptual level, the separation of concerns needs to address three issues. 1) Provide a sufficient abstraction for each concern as an individual concept. 2) Ensure that the individual concepts are primitive, in the sense that they address the natural concerns in the mind of the programmer.*

Implementation level. *At the implementation level, the separation of concerns needs to provide an adequate organization that isolates the concerns from each other. The goal at this level is to separate the blocks of code which address the different concerns, and provide for a loose coupling of them.*

The concerns identified at the conceptual level are mapped into the implementation level using a programming language. Separation of concerns at the conceptual level is generally considered a primary means to manage complexity in all engineering disciplines. However, few programming languages allow these abstractions to actually

Copyright © 2002 by the author. All rights reserved.

be separately programmed. The resulting code organization is monolithic, intertwining statements for different purposes.

We believe that programming all concerns in one monolithic program block increases complexity considerably and unnecessarily. By abstracting concerns out and separating them, programming individual concerns becomes substantially less complex, and code can be effectively reused.

We turned then to some approaches that had been suggested in the literature, including the work of our own group, Demeter (Lieberherr et al. 1994, Silva-Lepe et al. 1994). We identified a set of papers that focused on the separation of some concern from the basic algorithmic concern. Table 1 gives an overview of this survey; I have now added more references than what we originally had.

Technique → **Concern** ↓	*Meta-level Programming*	*Adaptive Programming*	*Composition Filters*	*Others*
Class organization		Lieberherr et al. 1994		
Process synchronization	Watanabe and Yonezawa 1990	Lopes and Lieberherr 1994	Aksit, Wakita et al. 1994	Frølund and Agha 1993 Reghizzi and Paratesi 1991
Location control	Okamura and Ishikawa 1994			Zeidler and Gerteis 1992 Takashio and Tokoro 1992
Real-time constraints			Aksit et al. 1994	Barbacci and Wing 1986
Others				Liskov and Schifler 1983 Jacobson 1986 Magee et al. 1989

Table 1. Approaches for separating certain concerns from the functionality of the programs.

2.3 Identifying Software Concerns That Can Be Separated

In the "Separation of Concerns" report, Walter and I then went on to analyzing the major software concerns that had been referenced in the literature. We focused on six: class organization, synchronization, location control (configuration issues), real-time constraints and failure recovery. We stressed the point that such as list was, by no means,

Copyright © 2002 by the author. All rights reserved.

exhaustive; we believed it was open-ended. For the purposes of that report, we were simply compiling a set of software engineering concerns that had been frequently referred to in different papers as problematic. We mentioned other examples such as debugging, persistence and transaction management.

In the years that followed, these concerns would be, again, the central focus of AOP-related work. Later on, AspectJ introduced general-purpose aspect programming constructs that made the concept of "aspect" more general and helped open up the list.

2.4 Separation Techniques

In the report, we made a distinction between separation of concerns at the conceptual level and at the implementation level. The former may exist without the latter, and that was pretty much the state-of -the-art in 1994. There were, however, some programming techniques that looked promising for achieving the separation at the implementation level. Table 1 shows the techniques we identified at the time. What follows are the highlights of our analysis.

2.4.1 Meta-Level Programming

Meta-level programming is a well-know paradigm that has been documented in several publications (Smith 1984, Maes 1987, Watanabe and Yonezawa 1990, Kiczales et al. 1991, Okamura and Ishikawa 1994, among others). A reflective system incorporates structures for representing itself. The basic constructs of the programming language, such as classes or object invocation, are described at the meta-level and can be extended or redefined by meta-programming. Each object is associated with a metaobject through a meta-link. The metaobject is responsible for the semantics of operations on the base object.

How does meta-level programming support the separation of concerns at the implementation level? *By trapping message sends and message receives to objects, metaobjects have the opportunity to perform work on behalf of the special purpose concerns. For example, they can check for synchronization constraints, assure real-time specifications, migrate parameters between machines, write logs, and so forth. This allows the base-level algorithms to be written without the special purpose concerns, which in turn can be programmed in the metaobjects. Also, by having structural reflection (meta-knowledge about the relations between classes), meta-level programming can achieve separation between algorithms and data organization.*

2.4.2 Adaptive Programming

The work described in Lieberherr et al. 1994 and Lopes and Lieberherr 1994 presents adaptive software, a programming model based on code patterns. The relations between the data structures of the application is described by graphs (called class dictionary graphs) to which the patterns apply. A pattern compiler takes a set of patterns and a class dictionary graph and produces an object-oriented program. Code patterns are classified in different categories, each one capturing abstractions in programming:

Propagation patterns *define operations (algorithms) on the data. Propagation patterns identify subgraphs of classes that interact for a specific operation. References to the data*

Copyright © 2002 by the author. All rights reserved.

are made in a structure-shy manner through succinct subgraph specifications, and the actual code is defined in code wrappers along traversal paths.

Transportation patterns *abstract the concept of parameterization. They are used within propagation patterns in order to carry parameters in and out along the subgraphs.*

Synchronization patterns *define synchronization schemes between the objects in concurrent applications. Their purpose is to control the processes' access to the execution of the operations.*

How does adaptive programming support the separation of concerns at the implementation level? *Each pattern category addresses a different concern. The patterns that define a program can be viewed as the basic software components that interact with each other in a very loose manner through name resolution. Each pattern is quasi independent of both the other patterns and the data organization with the effect that changes in the class organization don't necessarily imply updates in the operations, and modifications of the algorithms don't necessarily imply changes in the synchronization scheme.*

2.4.3 Composition Filters

The composition filter model is an extension of the conventional object-oriented model through the addition of object composition filters. For a detailed description of the model and its various applications we refer to Aksit et al. 1992, Bergmans 1994, Aksit et al. 1994. Filters are first class objects and thus are instances of filter classes. The purpose of filters is to manage and affect message sends and receives. In particular, a filter specifies conditions for message acceptance or rejection, and determines the appropriate resulting action. Filters are programmable on a per class basis. The system makes sure that a message is processed by the filters before the corresponding method is executed: once a message is received, it has to pass through a set of input filters, and before a message is sent, it has to pass through a set of output filters.

How do composition filters support the separation of concerns at the implementation level? *Separation of concerns is achieved by defining a filter class for each concern. For example, in Aksit et al. 1994 a real-time filter* `RealTime` *was proposed to affect the real-time aspects of incoming messages.* `RealTime` *filters have access to a time object that is carried with every message and which specifies the earliest starting time and a deadline for the message. Each filter class is responsible for handling all aspects of its associated concern. The filter mechanism gives programmers a chance to trap both message receives and sends, and to perform certain actions before the code of the method is actually executed. The resulting code is thus nicely separated into the special purpose concern (in the filter) and basic concern (in the method).*

Discussion

Common to the above techniques is the fact that they provide some mechanism to intercept message sends and receives. Metaobject protocols perform the interception at the meta-level through computational reflection and reification of messages. Composition filters trap messages through the built-in filter mechanism. In both cases, interception was done at run-time. Adaptive programming achieves message interception at compile

Copyright © 2002 by the author. All rights reserved.

time; the AP compiler detects when a method needs to be extended with code for special purpose concerns and inserts that code directly, i.e. a preprocessor.

An important aspect of meta-level programming is that the separation of concerns is not imposed by the model. Rather, meta-level programming facilitates the separation of concerns by providing the reflective information about the constructs of the language itself. Programming the special purpose concerns at the meta-level is a strategy that may or may not be followed by the programmers. This is contrary to filters and Adaptive programming, which provide specific language constructs to achieve the separation of concerns. A consequence of this fact is that in both the filters approach and the code patterns approach a new language construct is necessary for each new concern to be dealt with, while in the meta-level programming it is not so.

In retrospect, we missed at least one important piece of related work: Subject-Oriented Programming (Harrison and Ossher 1993). We also missed the opportunity to compare all these approaches with an emerging wave, design patterns (Gamma et al. 1994). But the important thing about our paper was to point out how the search for better expression mechanisms that focused on certain software development concerns were, in fact, driving a large number of research efforts at the time. This research was being driven by some common goal, and it was important for me to understand what that was; I wanted to formulate the kernel of the problem that was prompting so many solutions. Why weren't C++ or Lisp good enough?

3 The Birth of AOP @ PARC

In the summer of 1995, as I was starting to devise a thesis proposal based on some of these ideas, I went on an internship to Xerox PARC, in Gregor Kiczales' group. The group at the time was working on Open Implementations (Kiczales 1995, Kiczales 1996, Kiczales et al. ICSE 1997). During that summer I implemented Demeter's traversals in a dialect of Scheme that supported OO reflection (Lopes and Lieberherr 1996), reinforcing the idea that reflection was a powerful programming technique that could support Demeter's useful concepts for software evolution. Following that internship, I got an invitation to stay at PARC and continue my thesis work there. And so I did. The three years that followed were crucial both to the foundations of AOP and to me, personally: I defended my thesis at the end of the summer of 1997. Between 95 and 97, I continued to work under Karl Lieberherr's supervision, but I had Gregor Kiczales as a co-advisor.

I can't remember the exact date when we decided to call our work "Aspect-Oriented Programming," but I remember the term was suggested by Chris Maeda, the most business-oriented person of the group. Another name being tossed around was Aspectual Decomposition and Weaving (ADW), which was dropped. In my notebook, the first reference to "AOP" occurs at the end of November 1995. In January 1996, my notebook indicates that we were using Open Implementation and AOP at the same time, although for different pieces of the group's work. By June of 1996 we submitted a proposal to DARPA entitled "Aspect-Oriented Programming." By the end of 1996 the references to Open Implementation in my notebook disappeared.

One other word that defined AOP was the word "weaver." Again, I can't remember the exact date when that word emerged and who suggested it, but it must have happened in

Copyright © 2002 by the author. All rights reserved.

late 1995 or early 1996. Weaver was the name we gave to the pre-processors that would merge the components and aspect modules into base language source code. Later, this word was disfavored, because it had a strong connotation with text pre-processing. But "weaver" is still a good word for the AOP language processors, even as they are more than simple text pre-processing. The latest version of the AspectJ compiler is a good example of bytecode weaving that supports the join point model.

In October of 1996 we held a workshop at PARC to which we invited certain people who were pursuing work related to separation of concerns. That was the kick-off meeting for discussing AOP beyond our group; I'll say more about that in the next section. In this section, I'll focus on work done by the group at PARC.

3.1 RG

One of the projects going on at PARC when I got there was RG (Mahoney 1995, Mendhekar 1997). The concern that was targeted in that project was the optimization of memory usage when composing functions containing loops over matrices. Although the optimization of memory usage has never, since then, been analyzed as an aspect, the RG example was actually very interesting, and it was chosen as the leading example in the first AOP paper (Kiczales et al. ECOOP 1997). The reason why RG is interesting is that the problem in it illustrates quite well, even better than the AspectJ examples, what I think is the essence of AOP: the need for more powerful referencing mechanisms in a programming language. The aspects in RG expressed issues like the following (citing from Mendhekar et al. 1997):

> "For every message send invoking a primitive filter, before computing its arguments, examine each argument and determine whether the loop structure needed to calculate the filter is compatible with the loop structure needed to calculate the argument. In that case, generate a single loop structure that computes both the argument value and the filter value, and replace the original message send with a send to the fused loop."

While I might chose a slightly different wording, what this quote shows is that there is the need to refer to lots of different things: "every message send" of a certain kind, "before computing its arguments" and certain "loop structures" in the target object and the arguments. These are all referencing needs that are not supported by most programming languages, and that the group at PARC was trying to support.

3.2 AML

A second project under way was Annotated MatLab, or AML (Irwin et al. 1997). The problem addressed here was the optimization of certain MatLab programs, again focusing on memory usage and operation fusion. The AML solution was to annotate the MatLab code with special directives, mostly declarative, that augmented the code with information so that a language processor could produce optimized code.

There were many discussions among the group at PARC of whether AML was AOP or not. The final language annotations didn't look like our other systems, in that they weren't separated from the base code, they were still embedded in it. But, more

Copyright © 2002 by the author. All rights reserved.

importantly, it was hard to express in plain English the abstractions that those directives captured. For this reason, AML didn't make it to the ECOOP paper. It served, however, as a data point to formulate what aspect-oriented programming should(n't) be like.

3.3 ETCML

Between the summer of 1995 and the summer of 1997, John Lamping was working, among other things, on a little system called Evaluation Time Control Meta Language (ETCML). The idea was to provide a set of directives that programmers could use in order to instruct the language processor about when to evaluate certain pieces of code. This work was in the sequence of the work in Reflection, more precisely to identify whether certain parts of the code should be evaluated at compile-time or at run-time. This came from the need to optimize metaobject protocols, making them be compiled away. The thesis there was that the language processor could not always determine the best evaluation time, and that input from the programmer would simplify immensely the task of the language processor. In ETCML evaluation time was being analyzed as a software development concern that had important consequences on run-time performance.

This work served as another interesting data point to think about software development concerns that were relatively independent from the functional code.

3.4 DJ

Prior to the doctorate program, my background was in distributed operating systems (Sousa et al. 1993). That led me to the search for better expression mechanisms for distributed programming. When I went to PARC I had outlined my thesis in two publications: an ECOOP paper (Lopes and Lieberherr 1994) and an ISOTAS paper (Lopes 1996). Those were the pillars of my dissertation: a couple of small languages for distributed programming which I called D (as in Distributed Programming) and their specification as an extension to Java, DJ (Lopes 1998). The two little languages were called COOL and RIDL.[1]

DJ was different from RG, AML and ETCML, and used a technical approach more similar to that of Demeter (Lieberherr et al. 1994) than that used at the time by the group at PARC, i.e. Reflection (Kiczales 1991) and Open Implementation (Kiczales 1995, Kiczales 1996). For starters, DJ didn't target run-time optimizations; it targeted program-time expressiveness for some distributed programming concerns. RG, AML and ETCML had a top-down flavor: there was the notion of what a well designed program should look like and they were adding more instructions for tuning the performance without modifying the original well-designed programs. DJ had a bottom-up flavor: based on what distributed programs looked like, usually messy, I was trying to reorganize the code so that certain concerns that were tangled in Java could be untangled. In the process, I was defining language constructs that would allow me to do that. In the end, the combination of the top-down and bottom-up approaches proved to be fruitful.

[1] A few years later, in 2000 or so, my advisor Karl Lieberherr decided to rename "Demeter/Java" to "DJ" (Orleans and Lieberherr 2001, Lieberherr et al. 2001). Are you confused yet? Throughout this article, DJ refers to my DJ back in 1995-1997; my advisor's system will be called Demeter/Java, as it was at the time.

Copyright © 2002 by the author. All rights reserved.

I didn't particularly like the meta-level programming model. Certainly that model and the resulting techniques *could* be used to separate the concerns I was studying, but it felt awkward, albeit the only decent model at the time. Metaobjects have a beautiful run-time, interpretive semantics; I wanted a compile-time process. Compile-time reflection looses the beautiful simplicity of the run-time reflection model: metaobjects start to feel and act like macros. Therefore one is led to question whether that is the right model for compile-time processes at all. I didn't think so. I thought compile-time reflection introduced unnecessary complexity to the expression mechanisms I was looking for. Here is what I was looking for.

For synchronization, I wanted to be able to say things like "before executing the operation take in BoundedBuffer objects, make sure no other thread is executing it in the same object and make sure the buffer is not empty; otherwise wait" or "after executing the operation take in BoundedBuffer objects, check if the buffer is empty and, if so, mark it as empty; also, check if the buffer was previously full and, if so, mark it as not full". I also wanted to allow the expression of multi-object coordination schemes for concurrent agents like "before executing the operation activate in the Engine object, make sure the Door object is closed." It seemed awkward to me that in order to say this I would have to define metaclasses, instantiate and associate a metaobject for every base object, trap every message sent to the base objects and execute their metaobjects code at those points. For multi-object coordination schemes, the one-to-one association between base and meta objects wasn't even appropriate: we would want one single coordinator associated with the objects involved in the coordination scheme.

For remote parameter passing, I wanted to be able to say things like "when the operation getBook of Library objects is invoked remotely, the returned Book object should be copied back to the client, but the field shelfCopies shouldn't be included" or "when the operation borrowedBooks(User) of Library objects is invoked remotely, the only information that's needed from the User parameter is the User's name, so copy only that." Again, it seemed awkward that in order to express this I would have to use the reflection model.

Things would get even more confusing when these directives were to have a static code generation effect, which was what I was looking for. Although the reflection model might be a reasonable *implementation* model for the process, it certainly wasn't true to the intentions of synchronization and remote parameter passing directives, as expressed in plain English. The problem, then, was the expression of referencing.

So I came up with a simpler referencing mechanism, which was inspired by a body of previous work done by other people, but especially by my advisor Karl Lieberherr's Demeter system. The directives, expressed separately from the classes, would refer to the object's operations and internals by name:

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take};
  condition empty = true, full = false;
  put: requires !full;
       on_exit {
         if (empty) empty = false;
         if (usedSlots == capacity) full = true;
       }
```

Copyright © 2002 by the author. All rights reserved.

```
      take: requires !empty;
            on_exit {
              if (full) full = false;
              if (usedSlots == 0) empty = true;
            }
    }
```

and

```
    portal LibrarySystem {
      boolean registerUser(User user) {
        //Only strings. Everything else of User is excluded.
        user: copy {User only all.String;}
      };
      Book getBook(int isbn){
        //for the return object, exclude the field shelfCopies
        return: copy {Book bypass shelfCopies;}
      };
      BookList borrowedBooks(User user) {
        //for return object, exclude the field shelfCopies
        return: copy {Book bypass shelfCopies;}
        // for User, bring only the name
        user: copy {User only name;}
      };
    }
```

The binding, by direct naming, was unidirectional from these modules to the classes they referred to, and not the other way around. In other words, contrary to the dominating paradigm that said that each module must specify itself and its dependencies completely, this scheme allowed the definition of modules that would "impose" themselves on other modules, without an explicit request or permission from the latter. With this scheme, it was trivial to plug in and unplug concern-specific modules with a compilation switch.

This scheme also scaled nicely for multi-object schemes: just add more class names to the list of classes the coordinators and the portals were associated with, and we could refer to the operations and internals of those classes. E.g.

```
    coordinator Engine, Door {…}
```

3.5 DJava

Up until 1997, DJ was my own little piece of work, a system that I had carried with me from Northeastern, and one among others that we, as a group, were working on. In 1997 things changed.

I spent most of that year locked in my apartment writing my dissertation, so I didn't participate much in the group's activities. In the Spring, Gregor decided to invest the group's resources into the implementation of a DJ weaver, a pre-processor written in Lisp. That first language implementation, called DJava, supported COOL and some of RIDL. Over that summer, they planned a usability study. The users were four summer interns. They wrote a distributed space war game with it. This sure came handy as I was writing the Validation chapter! A report of those activities can be found in my dissertation (Lopes 1998, chapter 5). We used that application as an example for a long time.

At the end of the summer, Gregor decided to use DJava as the flagship system, the seed of what later became AspectJ.

Copyright © 2002 by the author. All rights reserved.

In the meantime, back at Northeastern, Karl Lieberherr also decided to incorporate DJ into Demeter/Java. That happened from the end of 1997 throughout 1998.

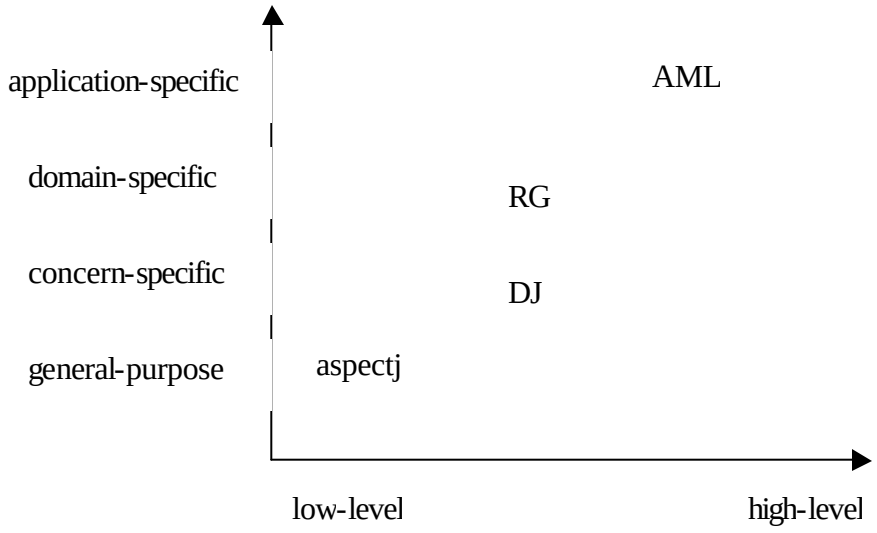
3.6 AspectJ

The first version of AspectJ, made public in March of 1998, was a reimplementation of DJava. It supported only COOL. Another release followed soon, I believe it was AspectJ 0.1. It included RIDL. A group at the University of British Columbia did some preliminary usability tests with this version. The results can be found in Walker et al. 1999.

As release 0.1 was coming out, at the end of April of 1998, AspectJ suffered a transfiguration. Gregor wanted to develop a general-purpose aspect language. DJ was a couple of concern-specific languages; it wasn't very useful for general purpose programming. The decision to make AspectJ general-purpose wasn't simple, at least for me. First, we had already released two versions, and changing the language's philosophy would probably confuse those who had been using it as a reference for AOP. But, most importantly, it wasn't at all obvious to me how a general-purpose aspect language would, indeed, be useful at the time, given the limited number of crosscutting concerns we had previously identified. What examples would we use to justify and explain it?

In retrospect, it is clear my fears were inconsequential. AspectJ is a lot more useful for a larger number of software development needs than it would have been if we had continued the path we initially set, which was, by design, limited. DJ served AOP well, but it was time to grow it. What follows is a brief analysis of what it took to make the shift from concern-specific to general-purpose.

In a paper we published in the summer of 1998 (Lopes and Kiczales 1998) we used the following picture to describe the range of languages we had been designing:



Copyright © 2002 by the author. All rights reserved.

How did we move from concern-specific to general purpose? What was preserved, what was added and what was thrown away? This is my view about the transition process.

Significant differences:

- The concept of having coordinators and portals as first-order elements of the language went away. AspectJ has "aspects." Aspects could, then, be coordinators and, eventually portals too. In fact, the subsequent releases of AspectJ had examples of aspects acting as coordinators and even reusable coordinator library aspects, which, because of the elimination of syntax, had a lot more lines of code than their DJ counterparts. But the good thing was that aspects could play lots of other roles without having to add more syntax. This was the design change that made AspectJ general-purpose.
- Central to DJ and Demeter was the concept of programming crosscutting concerns separately from the "primary" concerns, using special kinds of modules that could not be referenced back by the objects. The existence of aspect instances, and the possibility of their being handled in programs, was a point of much discussion, and during the first 2+ years of development we went back and forth on this issue (I call it "the metaobject syndrome"). The compromise was to use a singleton aspect instance by default. This is still the policy in the latest version of AspectJ, although it now provides a richer set of aspect instance associations. Unlike DJ, AspectJ provides handlers to aspect instances through the aspectOf() operation.

Significant clarifications:

- The concept of join point, which had been identified in DJ, RG and other systems, was cleaned up. DJ had only two kinds of join points: the reception of messages by objects (in COOL and RIDL) and the sending of messages to objects (in RIDL). Gregor envisioned a much richer set of join points that are now part of the AspectJ join point model. This extension, by itself, didn't make AspectJ general-purpose, but it certainly expanded the kinds of crosscuts it could express. In particular, this clarification allowed for the definition of control flow pointcuts, in later versions of AspectJ.

Significant preservations:

- Two basic principles were preserved: the presentation of AspectJ as an extension to Java and the implementation of the weaver as a compile-time process. Up until recently, the weaver was a pre-processor, transforming AspectJ programs into Java programs. Now it operates on bytecodes.
- Central to DJ and Demeter was the concept of referring to the join points using a very simple direct naming scheme based on the names of the classes and the fields. Since in DJ the "aspect" modules could refer to several classes, it used qualified names such as ClassName.FieldName, which could include wild cards. One design point was very important: *there was no notion of this naming process being a reflective operation.* To understand this, we have to look at the alternatives. In other languages, e.g. CLOS (Steele 1990), once we have an object, we can get the names of classes and members through a reflective API, and we can build meta-programs with that. DJ did not have the base-meta distinction and the API that goes with it; it had a simple declarative

Copyright © 2002 by the author. All rights reserved.

form for naming join points. That was preserved in AspectJ. The denotation of join points suffered several syntactic changes over the years, especially as we started to extend the kinds of join points supported by AspectJ. But unlike meta-programming, the naming is not programmatic but declarative: therefore, it feels very "natural," as declarative programming usually does.

- The temporal referencing before/after associated with join points existed in DJ and was preserved in AspectJ. (Note that before/after existed in other systems prior to DJ, namely in CLOS and in Demeter)
- The static introduction of structure and behavior had been defined early on for COOL (Lopes and Lieberherr 1994). CLOS (Steele 1990) also supported a similar feature, but for run-time. Introduction generated much discussion, as it didn't fit too well the run-time semantics of join points, but it was preserved in AspectJ. Over the years, it suffered several changes and clarifications.

Past the transition from concern-specific to general-purpose aspect language, which happened in 1998, AspectJ evolved considerably. Part of that evolution was due to the commitment to a solid advanced development plan. The support from DARPA, starting in 1998, allowed Gregor to get the resources he needed. In early 1999 the weaver was rewritten in Java, which made the system much more portable than the previous Lisp version. At the end of that year, there were extensions to existing Integrated Development Environments. The design of AspectJ stabilized when it got to release 0.7, in the first half of 2000. That was also the time I started pursuing other interests.

In form of conclusion to this section, it should be noted that lots of people were directly involved in the AOP project at PARC, at different times, besides Gregor Kiczales and myself. In the early days of AOP, the group included John Lamping, Anurag Mendhekar, Chris Maeda, Jean-Marc Loingtier and John Irwin. Venkatesh Choppella was there during the transition from DJ to the general-purpose AspectJ. Jim Hugunin and Mik Kersten joined in the transition to advanced development. Others, including Erik Hilsdale, joined after I left the project, and helped solidify the technology even further. Over the years, more than a dozen summer students contributed to the project; I can't remember all their names, so I leave them nameless.

4 Building Communities

Communities rarely happen spontaneously. It takes time and planning to create and expand them. The AOP group at PARC has put a significant effort in building communities around the technology. Lots of people outside our group were instrumental in helping clarify the concepts, by providing alternative technologies and all sorts of feedback. There were/are two kinds of communities: researchers and practitioners. The bridge was done by very special people: early adopters, those people who work in industry but have the curiosity and the will to try out beta systems.

4.1 Researchers

As mentioned earlier in the paper, in October of 1996 we held a workshop at PARC to which we invited researchers we knew were working in similar things. There were about 15 people in that meeting. The goal of that workshop was to discuss the major

Copyright © 2002 by the author. All rights reserved.

characteristics of, and compare, the work we all were doing. That included AOP (i.e. the PARC people), Subject-Oriented Programming (Ossher and Tarr), Composition Filters (Aksit and Bergmans), Reflection (Matsuoka et al.) and Adaptive Programming (Lieberherr). It was a fruitful workshop. One of the outcomes was the plan for a larger workshop associated with ECOOP'97, with the title "Aspect-Oriented Programming." That workshop attracted over 40 people and was a big success. AOP felt like the new kid on the OOP block. After that, there was an AOP workshop at ECOOP every year until 2000, and one AOP workshop at ICSE'98. At every workshop, I always met new people whose work would fit and enrich the separation of concerns/AOP umbrella.

4.2 Practitioners

Building communities of users, especially the "real" ones, is much harder than building communities of researchers. By "real" users I mean software engineers developing products in companies. Researchers thrive on ideas; practitioners thrive on solid systems that solve their problems without introducing new problems. Nobody in industry will use a system just because it embodies an interesting idea that will potentially help them.

Our first users were graduate students linked to the research community. They were the only ones who were motivated enough to skip through all the bugs! They weren't really using the language to build anything; they were using it as a reference point. Our first "real" users started to show up in the beginning of 2000. At this point, the compiler was solid enough to handle a few hundred classes. The first users who contacted us had read about AOP, had played a bit with the examples in AspectJ and wanted to try it in parts of their projects, with our support, for debugging aspects.

Early adopters are essential but they are also hard to deal with. They try something and they either like it – pushing it to the limit and asking for more – or drop it – silently. A handful of early users were patient enough to point out defects and weaknesses, and persisted in using AspectJ until it got a lot more solid. The vast majority were put off by the beta-ness of the language. Given that I left the AOP project later that year, I can't say much about what happened next. Accounting from the traffic in the mailing list, the articles in industry magazines and the third-party IDE support, it looks like AspectJ has been embraced by a large community. Some of the AOP ideas are here to stay!

5 Looking Back

It is quite interesting to look back to the period 1994-1997 and to compare my vision of AOP at the time with what AOP is now. My notion of Aspects[2] was based on systems I had worked on or studied. So, back then, according to my "Separation of Concerns" report, Aspects, independent of the techniques used to program them, were things like synchronization, remote parameter passing, configuration issues, real-time constraints, object structure, failure handling, persistence, security, debugging, etc. When I went to PARC, I found out about run-time performance Aspects such as memory optimization, loop fusion and evaluation time. Recently, I did a quick survey at what users are using AspectJ for, by looking at articles in industry magazines (Spurlin 2002, Grosso 2002,

[2] I am using Aspect with a capital A to denote crosscutting concerns at the conceptual (design) level, not at the implementation (AspectJ) level.

Copyright © 2002 by the author. All rights reserved.

Lesiecki 2002, Laddad 2002) and posting a question in the users list. The following categorization is an attempt at organizing my findings:

(1) debugging and instrumentation Aspects such as tracing, logging, testing, profiling, monitoring and asserting. Most of the usages fall into this category. But some usages are very sophisticated. For example, one user reported having built a "virtual internal information bus" inside their application.

(2) program construction Aspects such as mixins, multiple-inheritance (e.g. for bean construction) and views;

(3) configuration Aspects such as managing the specifics of using different platforms and choosing appropriate name spaces for property management;

(4) enforcement and verification Aspects such as making sure the types of a framework are used appropriately, components' contract validation and ensuring best programming practices;

(5) operating Aspects such as synchronization, caching, persistence, transaction management, security and load balancing;

(6) failure handling Aspects such as redirecting a failed call to a different service;

The ability to use aspects as add-ons over classes, as well as to plug in and unplug different aspects with a compilation switch, is being perceived as the major advantage of AspectJ/AOP.

In retrospect, although we missed a few kinds of Aspects and mentioned a couple that didn't yet emerge in practice, the analysis that Walter and I made back in 1994, which was voicing a trend that was in the air, was a self-fulfilling prophecy! It is actually quite amazing that later we, at PARC, were able to design a language that supports this diversity of crosscutting concerns… *with just a few key concepts*. In other words: the path I had started on – the design of concern-specific languages – wouldn't scale!

Another interesting observation is that AspectJ does not support any of the run-time performance Aspects that the group at PARC was focusing on before I joined. This doesn't mean that those Aspects are irrelevant; it simply means that AspectJ doesn't provide the kinds of referencing mechanisms that are necessary to support them.

One last comment on whether the broad AOP thesis – i.e. that it leads to *better* programs – has been validated or not. I don't have enough data to be able to draw any scientific conclusion. My recent poking at the AspectJ users gave me anecdotal evidence, as some users described their systems and commented on their positive experiences. From where I stand now, which is relatively far from where I used to be, I can see that AOP is extremely popular. Maybe the academic thesis doesn't matter!, as it never mattered for all other languages (Lisp, C++, Java, etc.), and as long as AOP helps solving some practical software problems.

6 The Essence of AOP and Future Challenges

What is it about Aspects that makes them both attractive to researchers and useful to practitioners? And where can we go from here? I haven't worked in AOP for a couple of

Copyright © 2002 by the author. All rights reserved.

years, but being as fascinated by languages as I was then, it's very interesting to try to answer these questions.

First of all, programming languages are incredibly restrictive programming systems. They all have one fundamental weakness. They emphasize the fact that they are a means to define computational processes and they ignore the fact that they are a means for *humans* to write down, and read, computational processes. Humans don't think using any of the existing programming languages. Even if we do, we certainly haven't been writing down structured ideas for thousands of years using those languages. We have been using natural languages. That has worked out quite well. Natural languages are as general-purpose as languages can get. They contain an extremely rich and diverse set of constructs that allow us to write down and enormous amount of ideas concisely and in modular ways that can be easily understood by others.

Computer systems, of course, are different. I am not suggesting that programming languages should have a natural language interface. That has been suggested a long time ago (e.g. Sammet 1966, Ballard and Biemann 1979) and it has been done before (e.g. Hypertalk (Winkler et al. 1994) and NaturalJava (Price et al. 2000)); the result is always limited or dubious. However, I am suggesting that programming language designers should pay more attention to the way natural languages work and the way we structure ideas with them. This is related to what I think is the major contribution of AOP to the next generation of programming systems.

Take tracing, for example. When we think of tracing we formulate something like this: "for all methods, call Trace.in before they start executing and Trace.out after they finish executing." However, all programming languages will force us to transform this sentence into something like this: "In method A, call Trace.in; … call Trace.out; return. In method B, *etc*." So what is it about the first representation of the intention that's better than the second, and how does the natural language help? In this case it's the references to "all methods", "before … executing" and "after … executing". *That's the power of AspectJ: it supports a richer set of structural and temporal referencing that follows what we have in natural languages*. AspectJ does it in a way that seems to be very useful for practitioners: it allows the encapsulation of these forms in modules that can be added to or removed from the applications with a compilation switch. In other words, writing a tracing aspect is like writing a different chapter, or section, in a book.

So, what makes an Aspect be an Aspect, before we even think of programming it with AspectJ? Given the name we chose for it, which clearly influences our perception, Aspects are software concerns that affect what happens in the Objects but that are more concise, intelligible and manageable when written as separate chapters of the imaginary book that describes the application. This pseudo-definition of Aspect aligns well with what users have been using AspectJ for. The structural and temporal referencing in AspectJ are essential mechanisms for achieving the separation between the Objects and those other concerns. Those mechanisms are also *natural*: we would use those kinds of referential relations if we were to write it in English or Portuguese. But the need for better referencing mechanisms doesn't end with what the word "Aspect" conveys.

On the way to future challenges, I'll do a brief incursion into Linguistics. Linguistics has been studying a large super-set of the constructs that AspectJ supports: referentiality

Copyright © 2002 by the author. All rights reserved.

between utterances – the subject matter of binding theory draws its roots from Chomsky's pioneering work. In Natural Languages, pronouns (e.g. this, that, it, her, which, etc.) are examples of such referential relations, but they are not the only ones. In general linguistics, referential dependence is studied regardless of morphological form, regardless of whether it is context-dependent or context-free and regardless of whether it is about objects or about time. For example, references can be: lists of nouns such as "The president, the cat, the resident and the hat"; constraints on nouns such as "colorless liquids"; temporal references such as "after reading the input stream"; and combinations of the above. Note that these are forms we use intuitively, that make texts very concise and that allow us to organize our ideas as optimally as we can. This very rich set of references is what allows us, for example, to divide specification manuals into chapters and sections that are related but loosely coupled; it is also what allows us to make a statement and add more to it at a later point. If we didn't have these referential forms we would, indeed, have a hard time communicating.

Programming languages support a very small set of referential relations. In particular, reflective references, groups and temporal references are, practically, inexistent. They can be simulated by combinations of computation and new nouns. And that's exactly one of the things that make programs much more complex than they should be: programmers have to express a rich set of referencing forms using a very small set of referencing forms. In the process, intentions get diluted and tangled.

The future of AOP will probably benefit from removing the word "Aspect" out of its name! What's important for the next generation of programming languages is the exploration of the rich set of referential relations we find in natural languages. That will allows us to appropriately implement pieces of program specification not only as separate chapters, but also as sections, paragraphs and even sentences, in a way that's much more *natural*; it will help avoid redundancy, temporary variables and all sorts of programming oddities. This is, of course, a challenge for language designers and I have only some fuzzy ideas about how those languages should look like. It seems to me that the conceptual framework that's available from Linguistics is an excellent framework for programming languages too.

Acknowledgements

Thanks to John Lamping for reading an earlier draft of this paper and pointing out some memory lapses and inconsistencies. Thanks also to Mik Kersten for proofreading the paper.

References

Aksit M., Bergmans L., and Vural S. 1992. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In O. Lehrman Madsen, editor, European Conference on Object-Oriented Programming (ECOOP), pages 372:396, Utrecht, The Netherlands, June/July 1992. Springer Verlag, Lecture Notes in Computer Science. Vol. 615.

Aksit M., Bosch J., van der Sterren W., and Bergmans L. 1994 Real-Time Specification Inheritance Anomalies and Real-Time Filters. In Mario Tokoro and Remo Pareschi, editors,

Copyright © 2002 by the author. All rights reserved.

European Conference onObject-Oriented Programming (ECOOP), pages 386:407, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. Vol. 821.

Aksit M., Wakita K., Bosch J., Bergmans L., and Yonezawa A. 1994. Abstracting Object Interactions using Composition-Filters. In M. Guerraoui, O. Nierstrasz, and M. Riveill, editors, Object-Based Distributed Processing. Springer Verlag, Lecture Notes in Computer Science, 1994.

Ballard, B. and Biemann, A. 1979. Programming in Natural Language: NLC as a Prototype. Proc. ACM/CSC-ER Annual Conference, 228-237.

Bergmans L. 1994. Composing Concurrent Objects. PhD thesis, University of Twente, Enschede, The Netherlands, July 1994.

Barbacci M. and Wing J. 1986. Specifying Functional and Timing Behavior for Real-Time Applications. Technical Report CMU/SEI-86-TR-4 ADA178769, Software Engineering Institute (Carnegie Mellon University), 1986.

Frølund S. and Agha G. 1993. A Language Framework for Multi-Object Coordination. In Oscar M. Nierstrasz, editor, European Conference onObject-Oriented Programming (ECOOP), pages 346-360, Kaiserslautern, Germany, July 1993. Springer Verlag, Lecture Notes in Computer Science. Vol. 707.

Gamma E., Helm R., Johnson R., and Vlissides J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading, MA, October 1994. ISBN 0-201-63361-2.

Grosso W. 2002. Aspect-Oriented Programming and AspectJ. In Dr. Dobbs Journal. August 2002. http://www.ddj.com/articles/2002/0208/

Harrison W. and Ossher H. 1993. Subject-oriented programming (a critique of pure objects). In proc. Object-Oriented Programming Systems Languages and Applications (OOPSLA), pp.411-428. 1993.

Honda Y. and Tokoro M. 1992. Soft Real-Time Programming through Reflection. In International Workshop on Reflection and Meta-Level Architecture, pages 12:23, Tama-City, Tokyo, Japan, November 1992.

Hürsch W. and Lopes C.V. 1995. Separation of Concerns. Northeastern University, College of Computer Science Technical Report NU-CCS-95-03. February 1995. ftp://ftp.ccs.neu.edu/pub/people/crista/publications/techrep95/index.html

Irwin, J., Loingtier, J.-M., Gilbert, J.R., Kiczales, G., Lamping, J., Mendhekar, A. and Shpeisman, T. 1997. Aspect-oriented programming of sparse matrix code. Scientific Computing in Object-Oriented Parallel Environments. First International Conference, ISCOPE 97. Proceedings. Springer-Verlag, 1997. p.249-56.

Jacobson I. 1986. Language Support for Changeable Large Real Time Systems. In Proc. Conference on Object-Oriented Programming Systems, Tools and Applications (OOPSLA'86). ACM Press. pp. 377-384.

Kiczales G., des Rivieres J., and Bobrow D.G. 1991. The Art of the Metaobject Protocol. The MIT Press, Cambridge, Massachusetts, 1991. ISBN 0-262-11158-6 (hc.).

Kiczales G. and Andreas Paepcke. 1995. Open Implementations and Metaobject Protocols. Tutorial slides and notes. http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf

Kiczales, G. 1996. Beyond the black box: open implementation. IEEE Software, vol.13, (no.1), IEEE, Jan. 1996.

Copyright © 2002 by the author. All rights reserved.

Kiczales G., Lamping J., Lopes C., Maeda C., Mendhekar A. and Murphy G. 1997. Open Implementation Design Guidelines. In Proc. 19th International Conference on Software Engineering (ICSE). ACM Press. 1997.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M. and Irwin J. Aspect-Oriented Programming. 1997. In Proc. 11th European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag LNCS 1241. June 1997.

Laddad R. 2002. I want my AOP! In Java World magazine. January, March and April 2002.

Lesiecki N. 2002. Test flexibility with AspectJ and mock objects. In Java Technology Zone for IBM's Developer Works. May 2002.

Lieberherr K.J., Silva-Lepe I., and Xiao C. 1994. Adaptive object-oriented programming using graph-based customization. Communications of the ACM, 37(5):94:101, May 1994.

Lieberherr K.J., Orleans D. and Ovlinger J. 2001. Aspect-Oriented Programming with Adaptive Methods. In Communications of the ACM 44(10). October 2001.

Liskov B. and Scheifler R. 1983. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Transactions on Programming Languages and Systems, 5(3):381:404, July 1983.

Lopes C.V. and Lieberherr K.J. 1994. Abstracting Process-to-Process Relations in concurrent Object-Oriented Applications. In Mario Tokoro and Remo Pareschi, editors, European Conference onObject-Oriented Programming (ECOOP), pages 81:99, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. Vol. 821.

Lopes C.V. and Lieberherr K.J. 1996. AP/S++: Case-study of a MOP for purposes of software evolution. Cristina Lopes and Karl Lieberherr. In Proc. Reflection'96, San Francisco, California. 1996.

Lopes C. 1996. Adaptive Parameter Passing. In Proc. International Symposium on Object Technologies for Advanced Software (ISOTAS'96). Springer-Verlag LNCS n.1049. Japan, 1996.

Lopes C. 1998. D: A Language Framework for Distributed Programming. PhD Thesis, College of Computer Science, Northeastern University.

Lopes C. and Kiczales G. 1998. Recent Developments in AspectJ. In Proc. Aspect-Oriented Programming Workshop at ECOOP'98. Workshop Reader, Springer-Verlag LNCS 1543. July 1998.

Maes P. 1987. Concepts and Experiments in Computational Re ection. In Norman Meyrowitz, editor, Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), pages 147{155, Orlando, Florida, October 1987. ACM Press. Special Issue of SIGPLAN Notices, Vol.22, No.12.

Magee J., Kramer J., and Sloman M. 1989. Constructing Distributed Systems in CONIC. IEEE Transactions on Software Engineering, 15(6):663:675, June 1989.

Mahoney J.V. 1995. Functional Visual Routines. Xerox Palo Alto Research Center Technical Report SPL95-069, July 1995.

Matsuoka S. and Yonezawa A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, Research Directions in Concurrent Object-Oriented Programming, chapter 1, pages 107:150. The MIT Press, Cambridge, Massachusetts, 1993.

Copyright © 2002 by the author. All rights reserved.

Mendhekar A., Kiczales G. and Lamping J. 1997. RG: A Case-Study for Aspect-Oriented Programming. Xerox Palo Alto Research Center Technical Report SPL97-009 P9710044. February 1997.

Okamura H. and Ishikawa Y. 1994. Object Location Control Using Meta-level Programming. In Mario Tokoro and Remo Pareschi, editors, European Conference on Object-Oriented Programming (ECOOP), pages 299:319, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. Vol. 821.

Orleans D. and Lieberherr K.J. 2001. DJ: Dynamic Adaptive Programming in Java. In Proc. Reflection 2001. Springer-Verlag.

Price D., Riloff E., Zachary J. and Harvey B. 2000. NaturalJava: A Natural Language Interface for Programming in Java. Proc. ACM Intelligent User Interfaces Conference.

Reghizzi C. S. and de Paratesi G.G. 1991. Definition of Reusable Concurrent Software Components. In Pierre America, editor, European Conference on Object-Oriented Programming (ECOOP), pages 148:166, Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science. Vol. 512.

Sammet, J. 1966. The Use of English as a Programming Language. Comm. ACM, 9(3), 228-230.

Silva-Lepe I., Hursch W., and Sullivan G. 1994. A Report on Demeter/C++. C++ Report, 6(2):24:30, February 1994.

Smith B.C. 1984. Reflection and Semantics in Lisp. In ACM Symposium on Principles of Programming Languages, pages 23:35, Salt Lake City, UT, January 1984. ACM Press.

Sousa P., Sequeira M., Ferreira P., Zúquete A., Lopes C., Pereira J., Guedes P. and Alves Marques J. 1993. Distribution and Persistence in the IK Platform: Overview and Evaluation. In Usenix Computing Systems Journal 6(4), Fall 1993.

Spurlin V. 2002. Aspect-Oriented Programming with Sun ONE Studio. In Sun ONE Studio Developer Resource page. October 2002. http://forte.sun.com/ffj/articles/aspectJ.html

Steele G. 1990. Common Lisp: The Language. Second Edition. Digital Press.

Takashio K. and Tokoro M. 1992. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems. In Andreas Paepcke, editor, Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), pages 276:294, Vancouver, Canada, October 1992. ACM Press.

Walker R.J., Baniassad E.L.A., Murphy G.C. An initial assessment of aspect-oriented programming. 1999. Proceedings of the 21st International Conference on Software Engineering (ICSE '99), Los Angeles, CA, USA, 16-22 May 1999. ACM, 1999. p.120-30.

Watanabe T. and Yonezawa A. 1990. Reflection in an Object-Oriented Concurrent Language. In Akinori Yonezawa, editor, ABCL: An Object-Oriented Concurrent System, chapter 3, pages 45-70. The MIT Press, Cambridge, Massachusetts, 1990. ISBN 0-262-24029-7.

Winkler D., Kamins S. and DeVoto J. 1994. Hypertalk 2.2: The Book. Random House.

Zeidler C. and Gerteis W. 1992. Distribution: Another Milestone of Application Management Issues. In G. Heeg, B. Magnusson, and B. Meyer, editors, Technology of Object-Oriented Languages and Systems (TOOLS Europe), pages 87-99, Dortmund, Germany, March 1992.

Copyright © 2002 by the author. All rights reserved.