Ph.D. Thesis Proposal: The Design and Implementation of Aspect-Oriented Languages

Doug Orleans

January 25, 2001

DRAFT — DO NOT DISTRIBUTE

1 Introduction

Aspect-oriented programming (AOP) [KLM+97] is a programming methodology gaining much attention in the software engineering field. It has the potential to increase code reuse, improve software maintainability, and shorten development time, by enabling the separation of concerns that are not easily separated using traditional methodologies such as object-oriented programming. AOP is not tied to any particular programming language, but most current work is focused on the development of AspectJ [K+], a general-purpose aspect-oriented extension of Java [GJS96]. Java is a wise choice for a language to extend; its popularity will make it easier to spread the idea of AOP, and it's a simple enough language to facilitate such extension. However, there are two main reasons to consider aspect-oriented extensions of other languages: to bring the benefits of AOP to users of other languages, and to make sure that the development of the AOP methodology isn't being shaped too much by the properties of Java. By thinking about AOP in the context of other languages, we might come up with fresh insights into the general nature of AOP, which might even pave the way for a formal semantics of AOP. Also, designing and implementing aspect-oriented extensions to other languages could lead to new design and implementation techniques that could apply to AspectJ or alternate aspect-oriented extensions to Java.

What languages should be considered for extension? There are lots to choose from. Rather than using specific languages, however, I plan to examine particular language *features*, separately and in combination. Considering a language feature individually will limit the variables involved, and will lead to a clearer understanding of the impact of that feature on an aspect-oriented language. Then when two features whose impact has been quantified are combined, the impact of the combination itself can be clearly analyzed as well. A number of aspect-oriented languages can then be constructed on demand from this building-block set of language features.

What language features, then, should be considered? My strategy is to concentrate on features which are generalizations of features in Java. In particular, the features I plan to study are multiple dispatch, intercessory reflection, and object-basedness. (These terms are defined and explained in later sections.) The main advantage of choosing generalizations of features of Java is that then any program written in AspectJ can be reimplemented in the more general aspect-oriented languages that will result.

My implementation strategy is to develop prototype languages by embedding them in Scheme [KCR98]. Scheme is well-suited for language development due to its syntax extension mechanisms. For example, a simple CLOS-like language with a meta-object protocol [KdRB91] called Tiny CLOS [Kic92] has been embedded in Scheme. In some cases it might be worthwhile to modify the AspectJ compiler `ajc` directly to handle a new language feature, but most of the time it will probably be better to have all the experiments working in the same environment, that of Scheme; in particular, that will make it easier to combine language features.

2 Aspect-Oriented Programming

Aspect-oriented programming is a way to achieve the benefits of separation of concerns when a concern is not easily separated using procedural or object-oriented decomposition. For example, suppose you want to add tracing statements to your program, for debugging: every method execution should print a message before it starts and after it finishes. The functionality of printing out messages, deciding what level of detail should be printed, keeping track of indentation, etc., can be encapsulated into a tracing module. But you would still have to add a call to the beginning and end of the body of every method in your system, in order to invoke the tracing module. These calls are scattered throughout the code, rather than being organized in one place; moreover, the body of every method is now tangled with these extra calls, which can distract from the purpose of that particular method. With AOP, instead of placing these calls in the body of every method, you would instead define an **aspect** which contained instructions for placing the calls, and combine the aspect with your program using an **aspect weaver**. The aspect could be written in a special-purpose tracing aspect language, or in a general-purpose aspect language (such as AspectJ). The former is easier for the aspect writer, since a program in a domain-specific language is usually more concise than in a general-purpose language, but more work for the aspect language implementor, especially if the aspect writer needs to do something that isn't possible in the aspect language and the language needs to be redesigned.

The benefits of this kind of separation of concerns (known in the AOP literature as **cross-cutting** concerns [KLM+97]) are the same as any separation of concerns: it makes software easier to understand, easier to change, and easier to reuse and adapt [OT00]. It does these by reducing scattering and tangling, by localizing all code relevant to one concern into one program structure.

How does AOP differ from other kinds of separation of concerns? One good litmus test for whether a language is aspect-oriented or not is whether the language has **quantification** and **obliviousness** [FF00]. Quantification is the ability to quantify a piece of code over other program elements (either static or dynamic); in other words, you can say "whenever condition C arises, perform action A." What can be specified in condition C depends on the degree of quantification in the AO language; it is generally on the order of "whenever method M is executed", but it could be as fine grained as "whenever variable v is accessed" [MHO96] or "whenever line L is executed" [Cla73]. Obliviousness is the ability to quantify code over programs that have been written oblivious to this code. In other words, a program does not have to explicitly cooperate with an aspect that refers to the program; in the example above, the code that's being traced does not need to be modified or otherwise written in any particular way in order to enable the tracing aspect to be implemented. A certain degree of obliviousness is essential for any code to be reusable, but ideally AOP can be used with code that was not intended to be reused at all.

3 AspectJ

AspectJ [K^+] is a general-purpose aspect language that is an extension of Java [GJS96]. An aspect definition is like a class definition with some extra parts: **pointcuts**, **advice**, and **introductions**. A pointcut defines where to weave, and advice and introductions define what to weave.

A pointcut specifies a set of **join points**, which are points in the dynamic execution of a Java program. For example, `executions(void f())` is a pointcut containing all executions of methods named `f` with no arguments returning void. `instanceof(C)` is a pointcut containing all join points that occur when the currently-executing object (i.e. the value of `this`) is an instance of class `C`. Pointcut designators can be compounded with boolean operators; thus `instanceof(C) && executions(void f())` is the intersection of the previous two pointcuts. Wildcards can be used in pointcut specifiers as well: `executions(* f(..))` contains all executions of methods named `f`, regardless of argument types or return types.

A piece of advice consists of a pointcut, a block of code, and an instruction for when to execute the code relative to the join points in the pointcut: `before` them, `after` them, or instead of them (`around`). In the case of a piece of `around` advice, the block of code can call the special function `proceed()` to execute the join point. The special variable `thisJoinPoint` is also available; it contains an object describing the current join point and can be queried with methods such as `getExecutingObject()` and `getSignature()`.

An introduction lexically inserts fields and methods into the text of a particular class. It can also add an interface to a class's `implements` list.

Here is a simple example of aspect definition, for our tracing example described above (a simplified version of the Trace example from the AspectJ primer [K^+]):

```
aspect Trace {
```

```
  void traceEntry(Signature s) { System.out.println("Entering " + s); }
  void traceExit(Signature s)  { System.out.println("Exiting " + s); }
  abstract pointcut classes();
  pointcut methods(): classes() && executions(* *(..));
  pointcut constructors(): classes() && executions(new(..));
  before(): methods() || constructors() {
    if (trace) traceEntry(thisJoinPoint.getSignature());
  }
  after(): methods() || constructors() {
    if (trace) traceExit(thisJoinPoint.getSignature());
  }
  introduction * { boolean trace; }
}
aspect TraceMyClasses extends Trace {
  pointcut classes(): instanceof(Foo || Bar || Baz || Garply);
}
```

Here, tracing can be turned on per-object by setting the `trace` field of that object to `true`. This aspect would be woven into the rest of the code by invoking the AspectJ compiler:

```
% ajc Trace.java TraceMyClasses.java Foo.java Bar.java Baz.java Garply.java
```

which produces `.class` files for the five classes. (`ajc` can also be run with the `-preprocess` option, generating new `.java` files into a separate directory.) Note also that the Trace aspect can be reused with a different set of classes by defining a new sub-aspect that defines the `classes()` pointcut differently.

4 Multiple Dispatch

A typical object-oriented language involves message sends and method invocations. A message is sent to an object, and a method is chosen to be invoked based on the dynamic (run-time) type of the object the message was sent to (the receiver). In Java, the static (compile-time) types of the argument expressions also plays a part in the method selection, but not the dynamic types. For example, suppose these classes are defined:

```
class P {
  void f(P p) { System.out.println("PP"); }
  void f(C c) { System.out.println("PC"); }
}
class C extends P {
  void f(P p) { System.out.println("CP"); }
  void f(C c) { System.out.println("CC"); }
}
```

and suppose we have the following variables declarations:

```
P p = new P(); C c = new C(); P c2 = new C();
```

Then the expressions `p.f(p)`, `p.f(c)`, `c.f(p)`, and `c.f(c)` would all print the expected things, namely `PP`, `PC`, `CP`, and `CC`, respectively. However, while `c2.f(p)` would print `CP`, because the receiver `c2` is an object of class `C`, `p.f(c2)` would print `PP`, because the static type of the `c2` variable is class `P`.

This behavior can surprise even the experienced Java programmer sometimes. A more symmetric solution is to use **multiple dispatch**, where the dynamic types of all the arguments play an equal part in method selection, instead of Java's single-dispatch mechanism. In essence, a message is sent to a sequence of objects, rather than a message with a sequence of arguments being sent to a single object. CLOS [Ste90] and Cecil [Cha93] are two languages that have multiple dispatch; MultiJava [CLCM00] extends Java with multiple dispatch. MultiJava allows methods to dynamically dispatch on other arguments in addition to the receiver; an argument can have both a static type and a dynamic type (called its **specializer**). In MultiJava, we could rewrite the above example thusly:

```
class P {
  void f(P p) { System.out.println("PP"); }
  void f(P@C c) { System.out.println("PC"); }
}
class C extends P {
  void f(P p) { System.out.println("CP"); }
  void f(P@C c) { System.out.println("CC"); }
}
```

Then both `p.f(c)` and `p.f(c2)` would print `PC`, and both `c.f(c)` and `c.f(c2)` would print `CC`.

The main thing that multiple dispatch adds that is important for an AOP language is more kinds of method signatures that can be mentioned in pointcut designators. In the above example, `executions(f(P))` would include all four methods, but there should be a way to include just the methods that specialize on `C`, such as `executions(f(@C))`. An alternate approach would be to generalize `instanceof(C)` to mean any join point where the receiver or any of the arguments are of class `C`; this extends the notion of "currently-executing object" to the set of objects that the message is sent to. Perhaps a better solution would be to add a new pointcut specifier `specializers(C)`, which included joinpoints where any of the arguments were of (dynamic) type `C`, or `specializers(C1,...,Cn)` where the sequence of argument types matches `C1,...,Cn`.

5 Intercessory Reflection

A language with **reflection** allows programmatic manipulation of program elements as data. In an object-oriented language, if the reflective data is also object-oriented, it's known as a **meta-object protocol (MOP)**. For example, in Java, classes, methods, and fields are objects, of type `Class`, `Method`, and `Field`, respectively; `o.getClass()` returns object `o`'s class, and `c.getMethods()` and `c.getFields()` return arrays of class `c`'s methods and fields, respectively. Also, `m.invoke()` invokes method object `m`, and `f.get()` returns the value stored in field object `f`. This is known as **introspective** reflection. Java also allows new classes to be created at run-time, with `ClassLoader.defineClass()`, but classes can't be redefined, and methods and fields can't be added or changed. Moreover, the Java programmer cannot create a new *kind* of class, a subclass of the `Class` class, in order to override methods of the default class behavior. This ability is known as **intercessory** reflection. Smalltalk [GR89] and CLOS [KdRB91] have intercessory MOPs, and OpenJava [TCKI00] extends Java with an intercessory MOP, although OpenJava's MOP is only compile-time, making it more like a macro-expander than full-fledged reflection.

The main impact of an intercessory MOP on an aspect language is the ability to modify classes at run-time. If a new method is added to a class that matches a pointcut defined in an aspect that's "in play", should the advice be hooked into that new method? Or is aspect attachment a one-time transformation? AspectJ already has to answer a related question: since it only generates code at compile-time, it can't anticipate new classes being defined at run-time. Böllert [Böl98] designed an aspect weaver for Smalltalk that replaces woven classes with generated subclasses, but doesn't detect modifications to already-woven classes. While it's possible to do this sort of on-demand weaving with a generative approach, an alternate way of implementing AOP in a language with an intercessory MOP is to make new kinds of classes and methods that call the appropriate advice at each join point. I've developed some meta-objects in Tiny CLOS [Kic92] to do this.

Tiny CLOS has multiple dispatch in the form of **generic functions**. Methods are attached to generic functions, rather than being attached to classes. For example, the methods named `f` in the previous section would all be methods attached to the generic function `f` and take two arguments rather than one:

```
(define-generic f)
(define-method (f (this <p>) (p <p>)) (display "PP"))
(define-method (f (this <p>) (c <c>)) (display "PC"))
(define-method (f (this <c>) (p <p>)) (display "CP"))
(define-method (f (this <c>) (c <c>)) (display "CC"))
```

I define a subclass of the class `<generic>` called `<aspectizable-generic>`; when an instance of it is applied to a list of arguments, all advice that is applicable is run before, after, or (in the case of `around` advice) instead of the corresponding points in the method selection and execution algorithm (with the continuation being passed to `around` advice).

In order to determine which advice is applicable, each piece of advice's pointcut needs to be compared to the current join point. Rather than try to duplicate AspectJ's pattern-matching style of pointcut specifiers, however, I decided to generalize pointcuts into arbitary boolean functions. To wit, a pointcut is a generic function that takes an aspect instance and a join point object (a descendant of the `<join-point>` class, such as `<execution-join-point>`) and returns true or false saying whether or not the pointcut contains the join point. (The reason it takes an aspect instance is so that pointcuts can be overridden by aspect subclasses; this has the side effect that the data stored as part of an aspect can play a part in the pointcut determination. I haven't yet come up with an example where a pointcut was relative to an aspect instance, but it seems potentially useful.) For example, the `classes()` and `methods()` pointcuts from the Trace example given earlier could be written like so:

```
(define-generic classes 'generic-class <pointcut>)
(define-generic methods 'generic-class <pointcut>)
(define-method (methods (trace <trace>) (jp <join-point>)) #f)
(define-method (methods (trace <trace>) (jp <execution-join-point>))
  (classes trace jp))
(define-method (classes (trace <trace-my-classes>) (jp <join-point>))
  (memq (class-name (car (method-specializers (slot-ref jp 'method))))
        '(<foo> <bar> <baz> <garply>)))
```

In other words, `(methods trace jp)` returns true if and only if `trace` is an instance of the `<trace-my-classes>` aspect, `jp` is an execution join point, and the name of the leftmost specializer of the method being executed is one of the classes being traced. This simulates the `instanceof` pointcut designator in AspectJ, by matching the "receiver" of the method; rather than checking for class equality, however, only the names are compared, which has the advantage of allowing this aspect to be defined *before* the classes to be traced are defined. Also, the wildcard matching of AspectJ could be achieved by comparing the name of the class (or the generic function) to a regular expression.

Once it is determined that a piece of advice is applicable to a join point, the advice is invoked. A piece of advice is, like a pointcut, a generic function; it takes an aspect instance and a join point (and a continuation, in the case of `around` advice) and executes the code in the advice. For example, the `before` advice from the tracing example could be written as:

```
(define-generic before-methods 'generic-class <before-advice>)
(define-method (before-methods (trace <trace>) (jp <join-point>))
  (if (slot-ref trace 'trace?)
      (trace-entry trace (get-signature jp))))
(slot-set! before-methods 'pointcut methods)
```

Since a piece of advice is a generic function, it could be overridden by sub-aspects, which isn't possible in AspectJ.

6 Object-basedness

Languages such as Self [UCCH91], Cecil [Cha93], and Obliq [Car94] are object-oriented, but do not have classes; they are known as **object-based** languages, or sometimes **prototype-based**, although that refers to a particular programming idiom that is not inherent in the language itself. Objects inherit directly from other objects, and in Self and Obliq, methods and fields are defined directly on objects themselves (in Cecil, methods and fields are defined on generic functions). When a message is sent to an object, if there is not a matching method defined on that object, the message is passed on to its parent object. Methods can be shared by multiple objects, by having the objects inherit from a parent object that holds the common methods. The *parent-of* relation thus serves the purpose of both the *class-of* and *superclass-of* relations in class-based languages.

Unifying these relations makes for a simpler language, without losing expressiveness: classes can easily be simulated in an object-based language. (Note that the reverse is less easy: to simulate an object-based language in a class-based language, every class would essentially need to be a singleton class, and object creation would involve class creation, which is typically expensive in a class-based language—if it can even be done at run-time at all.) An object-based aspect-oriented language would similarly be simpler but just as expressive. For example, there would be no distinction between aspect instances and sub-aspects. Also, pointcuts and advice would not need to be marked as static or non-static; static advice just doesn't rely on fields defined on an aspect having different values in the aspect's children objects. The `instanceof(C)` pointcut would just mean all join points where the receiver object has `C` as an ancestor.

7 Other language features

The three language features I have described above are the first three that I have considered, but there are others worth investigating as well. Two features that are already inherent in my Tiny CLOS implementation are **first-class functions** (i.e., functions may be assigned to variables, passed as arguments to other functions, and returned from functions) and **dynamic typing** (i.e., functions and variables are not annotated with their type). It remains to be determined exactly what impact these features has on an aspect-oriented language. One side effect of first-class functions is that functions (and classes) do not have an inherent name; they may be bound to different names in different environments. This might make join point specification harder, because it would have to be relative to a particular environment. Dynamic typing also makes join point specification harder, because there is less information about a function that can be used to differentiate it from others.

Another feature that would be interesting to consider is **predicate dispatching** [EKC98]. It generalizes multiple dispatch even further; rather than deciding what method to invoke based on the dynamic types of the arguments, a generic function can use any arbitrary predicate on the arguments to choose. For example, a generic function `pop` that takes a stack argument could choose to run different methods depending on whether the stack is empty or not. This fits in well with the notion that pointcuts are predicates; advice could just be methods on a generic function whose predicates are the pointcuts.

Other features worth looking at are **parametric polymorphism**, **multiple inheritance**, and **modules**.

8 Related work

Böllert [Böl98] designed a general-purpose aspect weaver for Smalltalk. It can incrementally weave aspects into classes, and remove them, at run-time. In order not to interfere with Smalltalk's development tools, which allow reflective access to the source code at run-time, source-to-source weaving is avoided; instead it uses Smalltalk's intercessory reflection to replace each woven class with a new class and to replace a woven class's metaclass with a new metaclass (to make sure that new instances are instances of the new class). It would be interesting to reproduce this approach in Tiny CLOS or OpenJava, and to compare with my even-less-invasive approach which does not involve replacing classes, but only adds a small (fixed) number of new metaclasses.

Several areas of research have similar aims to aspect-oriented programming, namely separation of concerns that are not easily separated using procedural or object-oriented decomposition. Some of them, in particular **subject-oriented programming (SOP)** [MHO96] and **composition filters** [AT98], have implementations in Smalltalk as well as Java and C++. SOP allows multiple class definitions with the same name to be automatically merged into one class; a **subject** is a collection of partial class definitions implementing a common concern, similar to an aspect. SOP has mostly been superceded by **hyperspaces** [OT00], which is a more general formulation of SOP allowing remodularization in multiple concern dimensions simultaneously. Currently the only hyperspaces implementation is HyperJ, which is a set of tools for Java. Composition filters allow message sends to be wrapped with multiple layers of filters, which are similar to aspects. Sina/st [Koo95] is a language embedded in Smalltalk that implements composition filters using reification of messages.

9 Contributions

The main contributions of my thesis research will be in an improved understanding of aspect-oriented programming untethered to any particular language. So far, specific contributions I have already tentatively identified include the idea of pointcuts being predicate functions; the ability to use aspect instance data in pointcuts; and the ability to override or extend advice in sub-aspects. These enhancements could all theoretically be applied back to AspectJ, but they arose by looking at AOP in the context of languages other than Java.

Specific implementation techniques are likely to be contributions of the research as well. In particular, the use of intercessory reflection to implement aspect attachment is a flexible but clean solution; the source-to-source translation approach used by AspectJ can be seen as an optimization of intercession, but studying a more dynamic form of intercession may lead to other ideas for optimizing while still keeping the flexibility (allowing for things like attaching and detaching aspects at run-time).

10 Research Plan

My research plan is to consider at least the three features detailed above, plus probably three or four features from section 7. Each feature should be considered in isolation or as close to it as possible, by considering a language only differing from the Java model with respect to that one feature. I've already gotten somewhat ahead of my plan by developing AOP for Tiny CLOS, which combines several features (namely multiple dispatch, intercession, dynamic typing, and higher-order functions). So the first step would be to develop an object model in Scheme that is similar to Java, and then sart modifying that in different directions, one feature at a time, considering AOP for each.

Where feasible, I would also like to migrate some of my AOP implementation ideas back to Java, or to extensions of Java such as MultiJava [CLCM00], OpenJava [TCKI00], GJ (Generic Java—Java with parametric polymorphism) [BOSW98], etc., by modifying the AspectJ implementation (which is open-source).

In order to demonstrate the benefit of using AOP with each of these language features, I need to provide some compelling examples of programs that become easier to express using AOP. To begin with, I plan to use some of the examples from the AspectJ primer; I have already started with the Trace example, and the Telecom simulation seems like another good example. However, I would also like to devise my own examples; one in particular that I've thought about is that of multi-user resource control, where each user's resource allocation (such as amount of memory, number of threads, or computation time) is limited by one or more resource policies, encoded as an aspect. My working model for this example is the JRes toolkit [CvE98], which uses a specialized Java class loader to rewrite bytecode class definitions in order to insert appropriate checks, in a somewhat aspect-oriented manner. I think this provides a good opportunity to show the benefits of AOP by using an aspect weaver rather than a class loader.

References

[AT98] Mehmet Aksit and Bedir Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. Technical report, TRESE project, University of Twente, Centre for Telematics and Information Technology, P.O. Box 217, 7500 AE, Enschede, The Netherlands, 1998. AOP'98 workshop position paper.

[Böl98] Kai Böllert. Aspect-Oriented Programming, Case Study: System Management Application. Graduation thesis, Fachhochschule Flensburg, 1998.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of 1998 ACM OOPSLA Conference*, Vancouver, October 1998.

[Car94] L. Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, June 1994.

[Cha93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, University of Washington, Department of Computer Science and Engineering, March 1993.

[Cla73] R. Lawrence Clark. A Linguistic Contribution to GOTO-less programming. *DATAMATION*, December 1973.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 1998 ACM OOPSLA Conference*, Vancouver, BC, October 1998.

[EKC98] Michael D. Ernst, Crag Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *Proceedings of ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20–24 1998.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, October 2000.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[GR89] Adele Goldberg and David Robson. *Smalltalk 80 The Language*. Addison-Wesley, 1989.

[K^+] Gregor Kiczales et al. AspectJ. Project web page. `http://aspectj.org`.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme, February 1998.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[Kic92] Gregor Kiczales. Tiny CLOS source code. OI software archives web page, December 1992. `http://www.parc.xerox.com/spl/projects/oi/software.html`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1997.

[Koo95] P. Koopmans. On the design and realization of the Sina compiler. MSc. thesis, Dept. of Computer Science, University of Twente, August 1995.

[MHO96] Hafedh Mili, William Harrison, and Harold Ossher. Supporting Subject-Oriented Programming in Smalltalk, August 1996. Presented at TOOLS USA 96.

[OT00] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Ste90] Guy L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

[TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-Based Macro System for Java. In Francesco Tisato (Eds.) Walter Cazzola, Robert J. Stroud, editor, *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.