Using Structure and Dependency Tracing Patterns for Aspect Composition

Uwe Zdun
New Media Lab, Department of Information Systems
Vienna University of Economics and BA, Austria
zdun@acm.org

Abstract

Aspects avoid tangled solutions for cross-cutting design concerns. However, there are various reasons why it may be hard to use an aspect language as a solution, even though developers are faced with cross-cutting design concerns. For instance, certain limitations of specific aspect composition (or weaving) mechanisms may hinder the use of aspects or the use is cumbersome. Or because of particular project requirements, such as used language, performance, or memory limitations, developers are not able to use an aspect language. In such cases, developers would benefit from better understanding current aspect composition mechanisms to customize existing techniques or implement (simple) aspect extensions from scratch. For these purposes, we present a pattern language for structure and dependency tracing, and then explain different, existing aspect implementations as sequences through this pattern language.

1 Introduction

This paper addresses possible implementation techniques for composing (or weaving) aspects. Different composition mechanisms and techniques for aspect-oriented software development (AOSD) [11] are distinct but yet comparable. The distinctive properties in the design of the AOSD implementations can be largely explained using a pattern language for tracing software structures and dependencies [24]. The patterns in this pattern language are often used in reengineering and software development tools. They provide solutions to find relevant architecture fragments in the source code, trace runtime structures, and inject structure and dependency changes. Note that we do *not* address a comparison of *aspect language concepts* in this paper, but only implementations of *aspect composition mechanisms* (in most cases these implementations are not even directly visible to the aspect language user).

After describing the pattern language briefly, we will concentrate on *different* implementations of AOSD composition mechanisms. These will be explained as sequences through the pattern language. First, we will explain generative aspect implementations, as used in popular AOSD extensions such as AspectJ [10] or HyperJ [20]. Secondly, we discuss dynamic message interception models, such as XOTcl message interceptors [16] or message interceptors in popular middleware (e.g. [9, 22]). Thirdly, we discuss a variant in which partial parsing techniques are used for implementing aspects. Finally, we discuss mixed approaches that introduce, for instance, more dynamics into generative aspect implementations.

A main motivation for this work is our observation that the term "aspect" is broader than the AOSD concepts currently realized with the techniques named above. These primarily realize extensional (or sometimes called "superimposed") aspects that can be separated in an (object-oriented) language construct (e.g. a class-like structure like an AspectJ aspect) and are executed for certain events in the method call flow. However, this kind of aspects is just one possible interpretation of the term "aspect" in the realm of software engineering. Design disciplines know other interpretations, and there is no reason to believe that other interpretations are less relevant for the software engineering discipline. For instance, Mørch sees aspect-orientation as a way to interweave the aspects design, programming, and use of software [14]. For example, in the context of reengineering it is often important to be able to separate such aspects or extract them from existing source code. Existing AOSD languages may help to architecturally separate some parts of these aspects, but such aspects can hardly be completely untangled. There are many other situations in which extensional aspects are not providing a complete solution: consider a situation in which an aspect should be added permanently to a system, as in many reengineering projects. Solving this problem with an extensional aspect is not an architecturally stable and clean solution, but program transformation is required.

Besides these requirements for conceptual additions to current AOSD concepts, there are also practical problems with given implementations that are recurring in many projects. Even though AOSD environments exist for many programming languages, there are still many language without AOSD support. If the computation environment is limited, as in embedded systems, it can be problematic to use current AOSD systems, as they produce some memory and performance overheads for their language runtime; however, from a conceptual point of view the aspect concept can be used to reduce or eliminate such overheads. For instance, the small components project [21] implements a project-specific aspect extension to avoid the overhead of a language runtime. Sometimes it is simply a business decision that no third-party language extensions should be used in a project. Some aspect models are already quite complex languages; for solving simplistic AOSD problems the required learning effort might be too large and writing a simple project-specific AOSD extension might be less effort.

In cases where existing AOSD extensions do not provide a suitable solution, developers either have to make additions or customizations to given implementations or implement their own (little, project-specific) AOSD extension. In this paper, we discuss patterns that are actually used for implementing aspect composition mechanisms in existing aspect languages or extensions. Note that the purpose is *not* to propose a new aspect concept, but to explain existing technical solutions, so that developers of new or customized aspect composition implementations can reuse the knowledge of the existing solutions. Of course, not only aspect language or extension developers benefit from understanding the implementation of aspect composition mechanisms: an aspect language user also requires a clear understanding of the consequences of the aspect composition mechanisms.

2 Understanding Structure and Dependency Tracing: A Pattern Language

Structure and dependency tracing techniques extract some knowledge from existing source documents of a software system or from the running system. We collectively refer to these pieces of information as *trace information*. The kinds of trace information required for a particular task may vary: for instance, for program transformation we usually require a full parse tree representation of the program text, whereas building an call graph requires dynamic interception of specific message flows. In the AOSD context, the goal of trace information extraction is to find the entities in focus of an aspect.

In Figure 1 a pattern language map is presented. For each individual pattern, we provide a pattern thumbnail with problem and solution in Table 1 (see [24] for more details).

The pattern language is used in many areas of software engineering. In the field of software maintenance and reengineering, the patterns are used for finding the structures and relationships in software systems. Typical maintenance and reengineering tasks that should be supported are code analysis, refactoring, visualization, metrics computation, tracing dependencies, analysis of quality attributes, grouping, integration, and wrapping. Reengineering tools, such as Rigi [15], support these tasks. Development tools (e.g. IDEs, profilers, architecture visualizations) also use structure and dependency tracing techniques. Programming language implementations and programming language extensions need to find existing structures and dependencies in the source code when parsing it. Component gluing and configuration mechanisms provide some means to compose software components in a customized way, either statically at compile time (or load time) or dynamically at runtime. At the composition time, the component composition mechanisms requires a knowledge about the (current) architecture configuration (for instance, to avoid loading required components twice). As aspect-oriented systems have to interpret and potentially manipulate either the software structures or the method call flow, they are often implemented with patterns of the pattern language (see next section).

3 Pattern Sequences for Aspect Composition Mechanisms

In this section, we explain current aspect composition techniques as sequences through the pattern language. As Alexander points out [1] pattern descriptions alone do not really allow a person to generate a good design, step by step, because they concentrate on the content of the patterns rather than laying the emphasis on morphological unfolding. The creative power lay in the *sequences* (or orders) in which the steps of applying patterns are to be performed. For a given task, the number of possible sequences is huge compared with the number of sequences which work, that is by comparison, tiny. Thus it is important to document the inherent knowledge in the pattern language in form of sequence examples that have proved to work in practice. Discussing such pattern sequences for the technical implementation of aspect composition mechanisms is the focus of this paper.

For any kind of extensional or superimposed aspect implementation, we first have to find the target entities, then change their behavior (or structure) according to the aspect definition, and finally execute the system with the aspect woven into the respective entities' implementation. There are many implementation variants for such aspects. In terms of the pattern language, these have in common that there has to be some way to extract the relevant pieces of trace information and modify the system accordingly. With different variants a language runtime can be built.

3.1 Using Program Generation to Weave Aspects

Currently the most common way to implement aspects are generative environments, such as AspectJ [10], HyperJ [20], D [13], ComposeJ [23] (a tool for adding composition filters [2] to Java), or JAC [18]. These follow a similar sequence. For illustration, we will give some examples from AspectJ. The other named generative environments use different terms and apply different transformations, but yet the basic implementation ideas are the same or similar.

The aspects as well as the points where to apply them are described in an extended language consisting of a set of additional instruction. This aspect language is added to the code written in the base language. Consider we have a Java class `Point` and assert certain properties using AspectJ. As additional statements, AspectJ introduces the `aspect` statement, as well as pointcuts (`call`, `target`, `&&`, etc.) and advices (`before`, `after`, `around`):

```
class Point  {...}
aspect PointAssertions {
  before(Point p, int x): target(p) && args(x)
      && call(void setX(int)) {
    if (x > 100 || x < 0) {
      System.out.println("Illegal value for x");
      return;
    }
}
```

The required trace information are the additional statements of the aspect language, the (Java) class and method structure, and the spots where invocations are sent or received (to handle the method call structures). A parses the program text and creates a parse tree. In AspectJ, the aspect language parser inherits from a Java parser, as the AspectJ syntax is an extended Java syntax. Note that an aspect language can potentially have a different syntax than the base language.

In a next step a injects hooks at the respective joinpoints (a process also called "inlining"). The aspect language code is replaced by base language primitives (or byte-code instructions of the virtual machine). The result is not visible to the user. The inserts hooks into the existing base language program code. These hooks call their implementations in this . The hooks together with respective implementations are a static form of . In the also other functionality of the language runtime is implemented (like the join-points). In AspectJ the implementation is realized as an advice; the injected hooks call the before, after, or around advices. Note that alternatively we could add (that for instance can trace variable accesses with operators or other internal structures of the). However, today's generative aspect languages concentrate on message interception.

At runtime of the woven program, the aspect language runtime is implemented as an (see Figure 2). AspectJ and the other named generative approaches implement a static weaving process. This means cannot be composed at runtime. Some tasks, however, are handled dynamically by the . For instance, AspectJ provides a dynamic join-point model. As a benefit, this allows pointcuts and advices to retrieve certain information about the message flow and the current join-point at

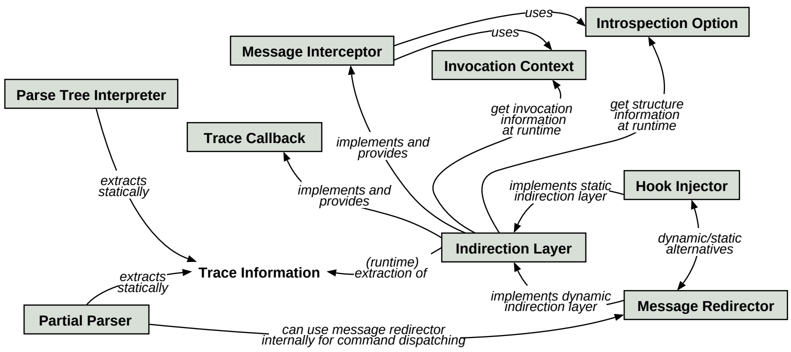

Figure 1. Important relationships of the patterns are represented by labeled arrows

Name	*Problem*	*Solution*
P -	Static trace information, as available from the code or other (formal) source documents, is required. How to extract (and possibly modify) the information in the source documents?	Parse the source language documents to create a parse tree and provide a tree traversal API. Use this API to build an application-specific that offers operations to extract (and modify) the trace information in the parse tree.
I -	Trace information can consist of information in source documents, but also of information derived from dynamic invocation data (and data flows). How to gather all relevant static and dynamic trace information in a unique way?	Provide an between the application logic and the sub-system that should be traced. The - wraps all accesses to the sub-system, and provides custom hooks to extract the relevant trace information. I is a generalization for other patterns, such as *object system layer* [7], *microkernel* [3], *virtual machine* [6], *interpreter* [5], and others.
T	You want to trace one or more specific structures of the runtime system. How to trace runtime structures of an - generically and dynamically?	Provide an interface to dynamically add or remove a for a specific runtime structure of the - . Whenever a specified callback event happens for the specified runtime structure, a user-defined callback operation is executed by the .
M -	An object-oriented intercepts and adapts all individual messages that are sent from the application logic to the hidden subsystem. How to gain control over the message flow in an object-oriented system so that we can at least trace (and modify) all messages and their results?	Provide a as a *facade* to the - . Application layer objects use symbolic (e.g. string-based) commands to access objects. The dispatches these invocations to the respective method and object.
H -	You do not want the trace code to be tangled within the sub-system code. Changes to trace code should be independent of the sub-system. How to trace (or modify) specific messages for a sub-system transparently?	Use a parser for the base language and inject the indirection hooks directly into the parse tree. Either write a custom compiler to directly create machine code or byte code, or, as a simpler alternative, produce a new program in the base language with the injected indirection hooks.
I -	Architectural structures of interest include dynamic structures (that can change at runtime) as well as static structures (that are defined at compile time and do not change at runtime). How to gather and provide such information?	All messages that are creating or changing structures or dependencies have to pass the . Offer - for each interesting architectural element. Provide a simple extension API for adding new, domain-specific .
I -	Invocation information are useful for building call graphs and object-oriented adaptations that rely on message exchanges. How to obtain the invocation information from inside of an invoked method or a wrapper method?	Provide access to the ' callstack by returning the current , an object (or other structure) representing the top-level callstack entry. The - contains at least information to identify the calling and called method, object, and class.
M -	How to dynamically compose message traces, modifications, or adaptations at runtime? How to reuse or refine message traces, modifications, or adaptations?	Provide in the that are dynamically composed with the invoked objects and operations. Before, after, or instead-of specified messages the are invoked.
	Sometimes some of the relevant information necessary to produce trace information are not yet documented or cannot be expressed with the given implementation language. How to add some trace information to a system?	Provide a standard notation for embedding in code or formal design documents. In these provide additional architectural knowledge and documentation (e.g. as hierarchical key/value lists).
P	A complete parser for the source document's language does not exist, it is too expensive, or it has bugs. How to access the trace information in source documents in a simple and extensible way?	Provide a simple parser that only understands the very basic syntax of the language. It only parses the specified language's subset, and ignores all other statements. For nested statements, the parser can be applied recursively.

Table 1. Patterns for structure and dependency tracing: pattern thumbnails

runtime. As a drawback, the dynamic parts of the language runtime consume additional runtime resources. Limited are offered for instance via the Java Reflection API.

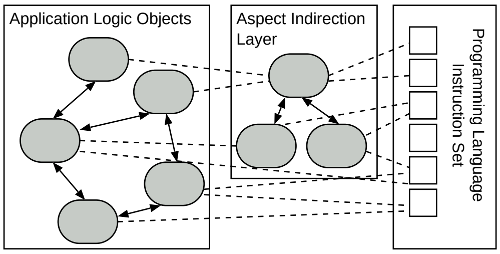

Figure 2. The aspect language runtime and advice implementations build an INDIRECTION LAYER

In AspectJ the original class implementation can also be extended with so-called introductions. For instance, in the example above it would make sense to have a method for checking the assertion, but this method requires the self reference of the current object. Thus it should be a method of the `Point` class:

```
aspect PointAssertions {
  private boolean Point.assertX(int x) {
    return (x <= 100 && x >= 0);
  }
  before(Point p, int x): target(p) && args(x)
      && call(void setX(int)) {
    if (!p.assertX) {
    ...
}
```

In a generative environment such introduction are implemented by injecting hooks into the respective classes.

The sequence for composing aspects generatively (perhaps together with a language runtime for dynamic tasks) is depicted in Figure 3. Note that the order of application is important.

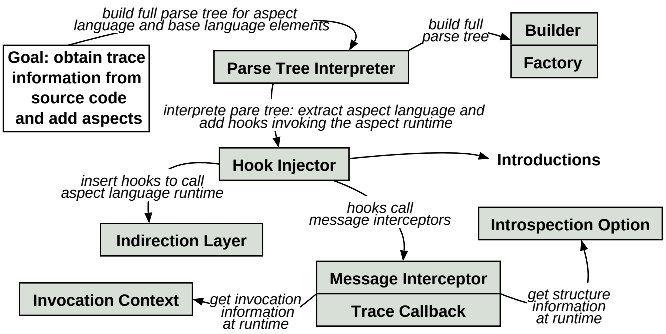

Figure 3. Generative aspect language with a language runtime sequence

3.2 Dynamic Implementations based on Message Redirection

Dynamic are provided by many different environments; especially by programming languages such as XOTcl [16] and by middleware environments, such as Orbix [9] or Tao [22] (also described in the *interceptor* pattern [19]). A can easily be implemented on its own; it just requires an in which messages can be dynamically intercepted. Typically a is used for implementing this layer. It receives symbolic invocations that are indirected to the actual implementations. A can dynamically intercept any message in the message flow, when it is dispatched by a . In particular:

- In the case of a programming language like XOTcl, the symbolic invocations are strings extracted from the program code. These invocations are indirected to the Tcl or XOTcl implementation (written in C), or other loaded components.
- In the case of a middleware the symbolic invocations are the remote calls that are sent across the network. On server-side there is a *server request handler* and an *invoker* that demarshal the invocation and dispatch it to the respective *remote object*.

In both cases, each invocation to a sub-system can be controlled by the that is added to the (see Figure 4). Thus each invocation in the application logic layer is evaluated using the . The maps the called symbolic instruction to a *command* in the . This *command* implements the base language instruction (in case of XOTcl, the *command*, for instance, implements an XOTcl object or operation).

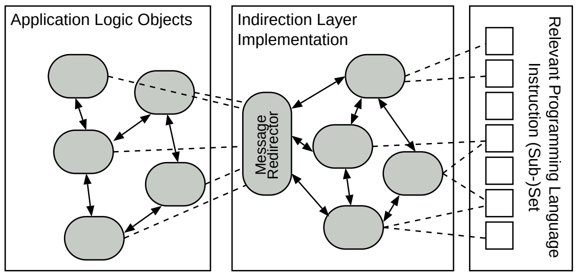

Figure 4. INDIRECTION LAYER architecture with a MESSAGE REDIRECTOR

The idea of applying aspects as dynamic on top of a (given) architecture is quite simple: if we specify all those calls that are in focus of an aspect as criteria for the , and let the execute this every time such messages are called, we can implement any aspect that relies on message exchanges. To receive the necessary information for dealing with the information, the should be able to obtain the (e.g. to find out which method was called on which object) and (to obtain structure information).

In case of Tcl also for variable slots are supported. That is, we can dynamically observe specified variables, when they are accessed in the .

For instance, XOTcl code corresponding to the above AspectJ point class example looks as follows:

```
Class Point
...
Class PointAssertions
PointAssertions instproc assertX x {
  if {$x <= 100 && $x >= 0} {return 0}
  return 1
}
PointAssertions instproc setX x {
  if {[my assertX $x]} {
    puts "Illegal value for x"
  } else {
    next
```

```
  }
}
Point instmixin PointAssertions
```

First, the corresponding code for the class and the aspect (here also implemented as a class) is defined. Then dynamically one of these classes is registered as an instance mixin for all points; thus all calls to the method `setX` are intercepted by the `PointAssertion` mixin's method `setX`.

In contrast to AspectJ, we do not have to "introduce" the method `assertX` on `Point`, as the mixin shares its object identity with the class it extends. However, in other cases we might want to change the class structure. In XOTcl at any time a new method can be defined. We may also dynamically check with an that it does not exist yet.

The *interceptor* pattern [19] is a variant of this implementation scheme, especially suitable for distributed environments, such as middleware platforms.

The dynamic trace information extraction sequence is depicted in Figure 5. Note that the order of application is important.

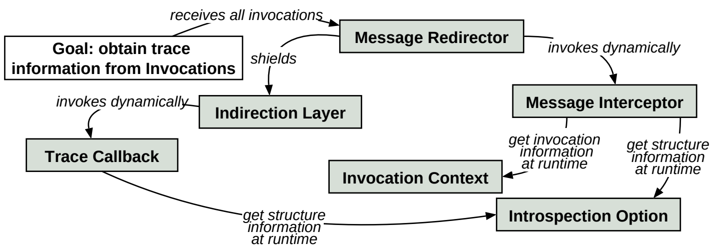

Figure 5. Dynamic trace information extraction sequence

DemeterJ [17] is a variant of the dynamic implementation technique that does not provide but uses traversal strategies and *Visitors* [5]. A class graph can be traversed as follows:

```
static final ClassGraph cg = new ClassGraph();
...
cg.traverse(this,
  "from Schema via ->TypeDef,attrs,* to Attribute",
  new Visitor() {
    void before(Attribute host) {
      if (host.name.equals("name"))
        def.add(host.value);
    }
  });
...
```

The `cg` object is a that first creates a (reusable) traversal graph and then the object structure is traversed. At each step in a traversal, the fields and methods of the current object, as well as methods on the *Visitor* object, are inspected and invoked by that are obtained via Java's Reflection API.

3.3 Partial Interpretation Aspects

In this section, we discuss an alternative implementation for aspects which we are currently developing within a framework, called Prowler, which is designed for dynamically extracting and transforming architecture information from a given program. Consider a situation in which we want to add an aspect to a system but have to get rid of some overhead produced by solutions using a or a - with a . Such situations can arise when the runtime impact or (slight) memory overhead of redirections or hooks is a problem. Or simply adding hooks or intercepting calls may be not enough, but more sophisticated transformations are required. Perhaps modifications should be added permanently and not as extensions. These situations are typical in a reengineering context.

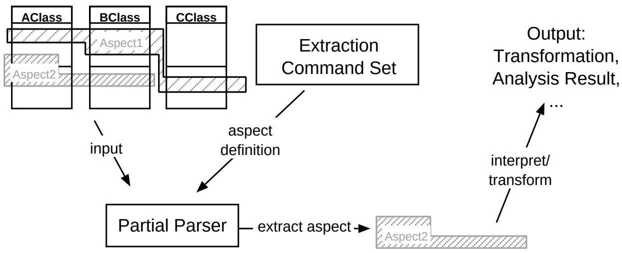

Figure 6. Aspects in architecture analysis or transformation with a PARTIAL PARSER

As a solution, we could use a for statically finding these information, but then possibly we would have to write a full parser for the language and this solutions cannot be used for extracting trace information from the running program. Extracting trace information using a can (in some cases) cause a significant performance impact and would require us to execute the system (often not suitable in a reengineering context).

P is another alternative for extraction (see Figure 6). As its input it reads a program with tangled aspects. It must be possible that these tangled aspects automatically can be found, say, because they are annotated in a comment. How to find these tangled aspects in the program text is specified in an extraction *command* set.

A addresses the problem that a complete parser for the source document's language does not exist or cannot (easily) be used. Often only a few language elements are interesting to create a certain architectural view, all others can be ignored. However, these spots are scattered across the whole source code. The output of the can be either interpreted programmatically or by a - .

The idea is to use a simplistic parser that only understands the very basic syntax of the language. It is enough, if it is able to find the beginning and the end of a statement and can read the statements one after another. In a we evaluate the statements and compare them to the extraction *command* set. If a *command* matches a statement, the *command* is invoked, and it extracts the required trace information from the statement (containing parts of the tangled aspect).

Note that the is not used for message interception in the program code but to implement the . The ' *commands*' actions are called only if the corresponding statement occurs in the program text. All statements that are not enlisted as *commands* in the - are ignored. The *commands'* actions are used to build up the relevant trace information.

3.4 Sequence Categories and Mixed Approaches

The sequences through the pattern language, as discussed in the previous sections, can be categorized regarding different criteria:

- the main trace information extraction pattern used: , , or ;

- aspects can be registered dynamically or are (statically) woven for the whole system;
- an aspect language runtime is supported or not;
- aspects support architecture modification or not.

The presented examples can rather cleanly be grouped into these categories. However, there are also some mixed implementation approaches.

For instance, the partial interpretation implementation, discussed in the previous section, primarily uses a that may feed a , if the output of the is still complex. However, we can also easily redirect the results of the to a - , say, by registering for the respective *commands* or by redefining their definitions. Thus we build a that creates a dynamic structure.

Another important variant are approaches that integrate benefits of the dynamic aspect composition into generative environments. For instance, the solution in [12] is a generative aspect model; however, it allows for activating and deactivating aspects at runtime, which is done via a central registry for aspects. This registry serves as a central for which every class is registered that contains a `superimpose` statement. A injects hooks into each method of these classes. The hooks call the registry in case of a method `dispatch`, `enter`, or `exit` event. If an corresponding - is registered as an advice, it is called by the registry before the original call.

Sometimes it makes sense to apply different aspect interpretations together. For instance, AJDC (AspectJ Design Checker) [8] is an extension of AspectJ that uses TyRuBa [4] as a logic meta-programming engine for finding errors and problems in AspectJ code. AJDC generates facts and rules out of the parse tree to be compiled and provides some rules for retrieving trace information like subclass relationships. Then TyRuBa is used to interpret this output. There are three additional *commands* understood by AJDC to define errors and problems in AspectJ code, and within them the TyRuBa syntax is embedded as a pointcut language.

The small components project [21] implements a projects-specific aspect composition mechanism (among other things), solely using generative techniques. The goal is to avoid overheads of an aspect language runtime in embedded systems.

4 Conclusion

We have described current AOSD implementation approaches using a pattern language for structure and dependency tracing. In this realm we believe the patterns capture the major implementation variants. We have only implicitly discussed the forces and consequences of the patterns. These mainly lead to the choice of appropriate patterns and pattern variants. Central, domain-specific issues like performance, flexibility, memory usage, program length, program complexity, etc. are highly different in different solution. Pattern language sequences were used to illustrate the existing solutions. The sequences should help developers to better understand existing AOSD language implementation choices. This understanding should enable developers to use, customize, or implement AOSD composition mechanisms.

Acknowledgements Thanks to Stefan Hanenberg for his helpful comments on this paper and to Markus Völter for discussion of the pattern language.

References

[1] C. Alexander and others. Patternlanguage.com. http://www.patternlanguage.com, 2001.

[2] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct 2001.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[4] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] J. Garcia-Martin and M. Sutil-Martin. Virtual machines and abstract compilers - towards a compiler pattern language. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.

[7] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.

[8] S. Hanenberg and R. Unland. Specifying aspect-oriented design constraints in AspectJ. In *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA 2002*, pages 641–655, Seattle, USA, Nov 2002.

[9] IONA Technologies Ltd. The orbix architecture, August 1993.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10), Oct 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[12] R. Lämmel, W. Lohmann, G. Riedewald, and C. Stenzel. Dynamically Registered Method-Call Interception via Source Code Instrumentation, 2002. submitted.

[13] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec 1997.

[14] A. I. Mørch. Aspect-oriented tailoring of object-oriented applications. In *Proceedings of the 21st Information System Research Seminar in Scandinavia (IRIS 21)*, pages 641–655, Aalborg University, Denmark, August 1998.

[15] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[16] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[17] D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, Sep 2001.

[18] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a flexible framework for AOP in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, Sep 2001.

[19] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[20] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, USA, May 1999.

[21] M. Voelter. Small Components Project, 2003. http://www.voelter.de/projects/smallComponents.html.

[22] N. Wang, K. Parameswaran, and D. C. Schmidt. Metaprogramming mechanisms for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio, TX, USA, Jan/Feb 2001.

[23] H. Wichman et al. ComposeJ Homepage, 2002. http://trese.cs.utwente.nl/prototypes/composeJ/.

[24] U. Zdun. Patterns of tracing software structures and dependencies. submitted to EuroPLoP 2003, a draft can be found at: http://wi.wu-wien.ac.at/∼uzdun/publications/archTracing.pdf, 2003.