The Proxy Inter-Type Declaration

Michael Eichberg
Software Technology Group,Dept. of Computer Science
Darmstadt University of Technology, Germany
eichberg@informatik.tu-darmstadt.de

ABSTRACT

Aspect-oriented programming [16] with its support for modularizing crosscutting concerns opens up the vision that component middleware can be replaced by sets of collaborating aspects that implement the infrastructural services. In this way, application servers could be customized on a per-project basis, as they are no longer monolithic applications.

In order for this vision to become reality it should be possible to implement services offered by current containers as aspects. In particular services targeting scalability and performance issues, such as passivation and instance pooling must be available. Such services build upon the concept of virtual instances [25]. They are realized in the current application servers by means of the (virtual) proxy pattern [11].

Implementing virtual instances represents a crosscutting concern, as we will discuss in the paper. Unfortunately, no current AOP language or framework has explicit support for modularizing this concern. We propose to support the generation of proxies as an inter-type declaration and argue that it is essential to allow the development of scalable and efficient component containers. It would be a significant improvement when compared with the current offered possibilities.

1. MOTIVATION

Aspect-oriented programming [16] with its support for modularizing crosscutting concerns opens up the vision that component middleware (e.g., application servers for Sun's EJB [9] or CORBA components [20] models) can be replaced by sets of collaborating aspects that implement the infrastructural services. Before this can be done the question arises how to achieve performance and scalability. Infrastructural services such as instance pooling and passivation of components are prominent solutions exploited by current application servers. The implementation of these services requires the use of *virtual instances* [25]: Instead of a component instance a virtual instance is exposed to the client (clients are all users of a component). To implement virtual instances the proxy design pattern [6, 11] is used [25]. One proxy object represents one virtual instance and the same proxy might refer to distinct physical component instances (one at a time) during its lifetime. All physical instances referred to by one (virtual) proxy object represent the same logical instance.

The concept of virtual instances is also used by the Enterprise JavaBeans component model [9]; the services which must be implemented by every EJB compliant container require that a component (an Enterprise JavaBean (EJB)) is not directly accessible; otherwise the services could be bypassed and the container would be compromised. Proxy objects automatically generated by the container, called `EJBObject`s, are the concrete realization of the virtual instances concept. An instance of an `EJBObject` always represents the same logical component regardless of its physical instance. It executes the explicitly requested services (in the deployment descriptor), such as security, transactions, etc., as well as implicitly provided services such as passivation, pooling, etc.

Although component passivation and pooling are services that span all components of the system in the same way and hence appear to be typical crosscutting concerns, we claim in this paper that it is not well supported by common AOP frameworks [3, 4] or AOP languages [15, 19]. Reports are indeed available about applying AspectJ to resource pooling, such as pooling JDBC connections [17,18]. However, pooling of resources is different from pooling of components in two significant ways:

1. A pooling aspect for resources is specialized to exactly one type of resource (e.g. database connections using JDBC) which is implemented by a fixed set of well-known classes. Thus, it is easy to determine the correct join points, i.e. to write the pointcut to be used by the aspect in order to integrate pooling. A pooling aspect for components, on the other hand, must be generic to handle multiple different types of components, e.g., different types of entity beans as different as `Order`, `Book`, `Customer`, etc., which are furthermore unknown at aspect development time. That is, we should be able to write the pooling functionality independently of the concrete type of the components that will be pooled. We will argue that this is not well supported by current mainstream AOP languages.
2. Pooled resources can be directly reused, as they are not client specific. On the contrary, when recycling physi-

cal components we need to keep their logical identity, so that the pooling is transparent to the client. So, if we want to reuse a pooled component, it is necessary to (re-)assign it the logical identity that is hold by the client. This forces us to integrate virtual instances in a consistent way all over the system.

To implement these services the ability to fully control all references to a component is required; a component can not be pooled as long as at least one other component has a direct reference to it that we cannot control. To implement the concept of virtual instances we can use proxy objects if and only if we simultaneously make sure that the proxy object can not be bypassed. Only under this condition is the proxy guaranteed to be the only one ever holding a reference to a component. The proxy can then be used as a virtual instance of a component.

To achieve the "non-bypassable" feature of a proxy the corresponding component needs to be transformed so that the reference to itself (`this`) is never passed to another component. Additionally, we have to check that `this` is not otherwise available e.g. via a public field. So, the following code:

```
1 class Order{
2   public void setCustomer(Customer c){ ...
3     c.addOrder(this);
4 } }
```

needs to be transformed into something similar to:

```
1 class Order{
2   public void setCustomer(Customer c){ ...
3     c.addOrder(getMyProxy());
4 } }
```

Also all component creations:

```
1     new Order();
```

needs to be transformed:

```
1     new OrderProxy(new Order());
```

This kind of transformation can be automatically performed if we can distinguish between components and normal classes. However, using standard pointcut and advice two solutions can be considered to simulate the effect of the transformation. The first "solution" would be to manually scan all component implementations and to write appropriate advice which passes the proxy instead of `this`. The second solution is to write one general advice which checks if a passed object is `this` and if so returns the proxy instead; given that the advice is generic, which means it simply advices all method calls to any component in the same way, it is necessary to check every passed parameter to every method call at runtime if the parameter is `this`. The effort to implement the first solution would be very high and also it would be tedious and error prone since an ignored statement that passes `this` to another component must not immediately result in a compile-time or runtime error. The inefficiency of the second solution would render any other optimization useless. So, the implementation of virtual instances is not well supported; the modularization of this cross-cutting concern with the current techniques offered by AOP frameworks or languages is hardly possible.

The reminder of this paper is organized as follows. In Sec. 2 we present our proposal by means of an example of a passivation service and indicate that what we are proposing is not well supported currently. In Sec. 3, we go into more technical details of the proposal. Then we discuss related work and conclude this position paper with a short summary.

2. THE DECLARE PROXY CONSTRUCT

We first give an overview of the syntax and semantics of the declare proxy construct by the example of the passivation service. Next, we go into some more details about the semantics of the proposed construct. Implementation details are out of scope for this position paper. In the following listings we use annotations and generics as they will be available in Java 1.5. However, we do not make any assumptions about their support in future AspectJ versions.

2.1 Overview of the Proposal

To support virtual instances, we propose a new construct for inter-type declarations in AspectJ: the `declare proxy` statement. The proposed syntax of the statement is as follows:

```
1 declare proxy
2   extends AProxy :
3   implements AnInterface;
```

The effect is that proxy classes are generated that extend the specified proxy class (line 2) and implement the specified interface (line 3). The interface (line 3) determines the components for which proxies are to be used: Proxies of type `AProxy` are generated for all classes that directly or indirectly implement `AnInterface`. This means the interface specified in the `implements` clause in line 3 has a selection functionality picking the set of components to be equipped with virtual instances, roughly comparable to a pointcut, and also determines the super-type of the proxies and components. The specified class (line 2) must not be final and must extend (directly or indirectly) the system type `Proxy`.

To illustrate the syntax and semantics of the `declare proxy` construct we consider its use within the implementation of a passivation service as shown in the snippet from the `Passivation` aspect below (line 2-4). Line 4 builds the bridge between the generated proxies and the components for which these proxies are to be used. On the one hand the interface determines the common super-type of the component and the proxy and on the other hand it determines for which components a `PassivationProxy` (line 2) is to be generated and used.

```
1 public aspect Passivation {
2   declare proxy
3     extends PassivationProxy :
4     implements SessionComponent;
```

To provide an accessible set of join points, the generated classes not only implement the specified interface, they also inherit a user supplied proxy class. In line 3 we additionally specify that the generated proxies have to inherit from the class `PassivationProxy`. This enables the definition of pointcut and advice in relation to a specific proxy and its subclasses and not only in relation to all classes that implement the specified interface.

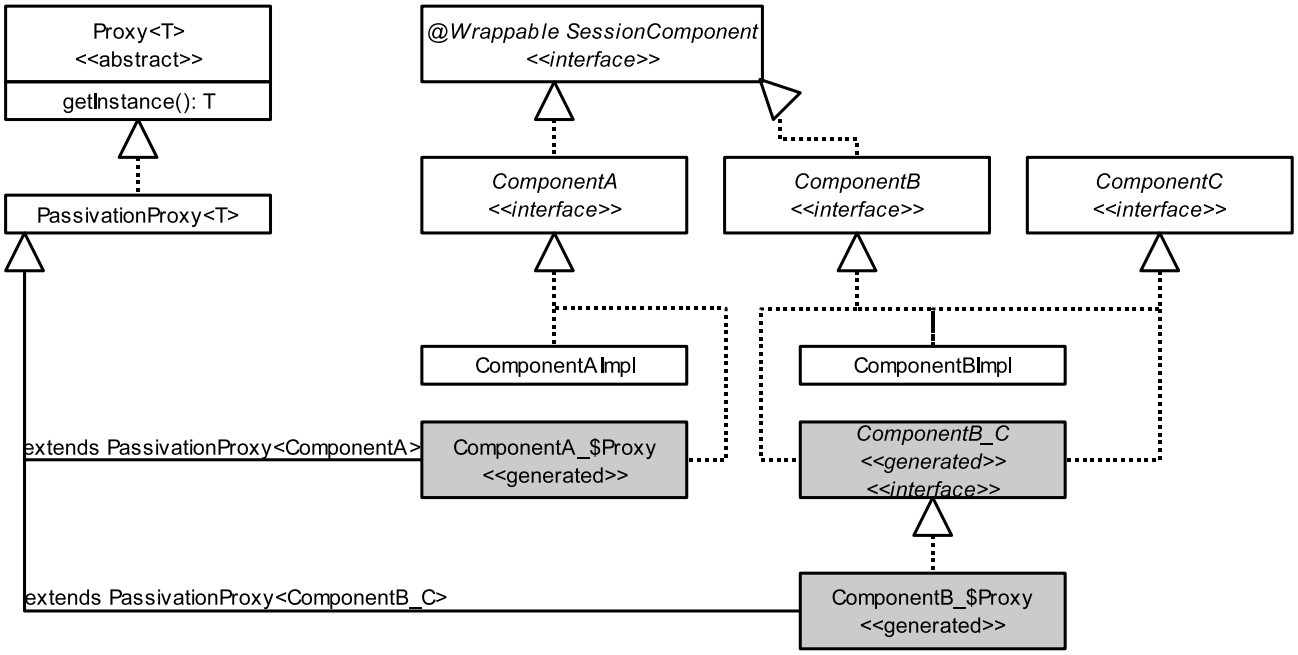

Figure 1: The effect of declare proxy: "`declare proxy extends PassivationProxy: implements SessionComponent;`"

Figure 1 shows the class diagram after the generation of the proxies, as the result of compiling the above declare proxy statement. For the component `ComponentAImpl` a proxy class (`ComponentA_$Proxy`) is generated that implements the interface of the component (`ComponentA`) and extends the specified `PassivationProxy`. Assuming that a component is always accessed via its interface, it is type safe to use a proxy instance instead of a component instance.

As seen in Figure 1, the proxy classes do not only implement the specified interface, they actually implement all interfaces of a component. The proxy class for `ComponentBImpl` must implement both interfaces (`ComponentB` and `ComponentC`) even though `ComponentC` does not extend `SessionComponent`. This is necessary to preserve the validity of the program. Code that casts between the component's interfaces is valid and must remain valid; this can only be assured if a proxy implements all interfaces. As a technical necessity, for every component which implements multiple interfaces an artificial interface has to be generated that extends all of them; this interface is then used to correctly bind the type parameter `T`. `T` determines the concrete interface type of a component (the interface `ComponentB_C` in Figure 1 is an example).

The interface (line 4) in the `implements` clause must be annotated `Wrappable` (see figure 1 for an example). For every class inheriting the `Wrappable` annotation proxy classes can be generated. The annotation with `Wrappable` enables us to distinguish between "normal" classes and components and to validate the later (see section on Implementation Restrictions). All component classes are transformed to no longer expose `this` unless no proxy is generated for the component. Further, it is assured that the program does not give rise to errors by the declaration and use of a simple forwarding proxy (calls to the proxy are directly forwarded to the corresponding method of a wrapped instance). To make this possible the interface type of a component is always to be used to access components. In addition, all created components instances are automatically wrapped by an associated proxy. This means every:
`new ComponentAImpl()`
expression is transformed into:
`new ComponentA_$Proxy(new ComponentAImpl())`.
This requires that all classes of a project needs to be recompiled whenever a `declare proxy` statement changes.

An important property of the generated proxy classes is that they are *anonymous*, Implying that we cannot write a pointcut that selects a joinpoint in a specific proxy (e.g. in `ComponentA_$Proxy`; the concrete type of a proxy is unavailable during the implementation of an aspect). This is due to the fact that the interface specified in the declare proxy statement does not directly correspond to one component; instead it determines a set of components that are related by the implementation of the interface. However, we can still refer to the proxies by their parent types as we will illustrate in the following by the advice of the passivation aspect.

Passivation (line 10) of components is executed asynchronously (line 3,15) if a component has been idle for the specified time (line 5, 8 and 9).

```
1  Set<PassivationProxy> proxies
2    = new WeakHashSet<PassivationProxy>();
3  { Thread thread = new Thread(
4      new Runnable(){
5        final long x = ...;
6        final long t = ...;
7        public void run(){
8          for (PassivationProxy proxy : proxies){
9            if (System.currentTimeMillis() −
10             proxy.lastAccess > t){
11           passivate(proxy);
12       } }
```

```
13              thread.sleep(x);
14            }
15          }
16        );
17        thread.start ();
18    }
```

In the default implementation of the proxies each method call is directly forwarded to the corresponding component method. Before this is done we want to make sure (as shown by the following code snippet) that the component is not passivated (lines 1-4) and after processing of the call by the component the time of the last access (lines 5-8) is updated.[1]

```
1   before (PassivationProxy proxy) : execution (..)
2     && !within(PassivationProxy) && this(proxy) {
3     if (isPassivated(proxy)) activate(proxy);
4   }
5   after (PassivationProxy proxy) : execution (..)
6     && !within(PassivationProxy) && this(proxy) {
7     proxy.lastAccess = System.currentTimeMillis();
8   }
```

The aspect additionally defines the methods to passivate and activate a component. The implementation details are omitted because they do not further contribute to the discussion of the `declare proxy` statement. The signatures are:

```
1   boolean isPassivated(PassivationProxy proxy){...}
2   void activate(PassivationProxy proxy){...}
3   void passivate(PassivationProxy proxy){...}
```

The last piece in the implementation of the `Passivation` aspect is the `PassivationProxy` class. In this implementation the proxy object is not only a virtual instance it also implements part of the logic of the service. The proxy class has a field to store the time of the last access (line 3) and also registers the proxy at the passivation service (line 6).

```
1  abstract class PassivationProxy<T>
2  extends DefaultProxy<T>{
3    long lastAccess;
4    public PassivationProxy(T instance) {
5      super(instance);
6      Passivation.aspectOf().proxies.add(proxy);
7    }
8  }
```

2.2 Proxies – Design Space and Decisions

Multiple possibilities exist for the generation of proxy classes. In the following we discuss them to determine the semantics of our declare proxy statement.

A proxy class can be (1) a subclass of a component or (2) an implementation of the interface of a component. The first possibility does not impose any particular requirements on the design of an application in order for it to be "decorated" by proxies; it simply requires that a component is not final, has no final methods and does not declare any non-private fields, otherwise the proxy could be bypassed. The problem with this alternative is that every instantiation of a proxy also instantiates the superclass (the component). So, a proxy object is in a sense also a component. This is not feasible if a proxy is supposed to be a (temporary) replacement of a component, e.g., to perform optimizations such as passivation, pooling and lazy initialization. The second solution assumes a particular design – every component must be implemented against an interface and the component's type is not allowed to be used directly (details are given later). However, this is only a small restriction since it is also considered good design to implement toward interfaces. Further, a proxy object is not a component, a proxy instance is fully independent from a component instance. So, we basically consider this solution the only feasible one. However, an equivalent alternative to the second solution is to require that every component fulfills implementation restrictions so that we are able to generate the necessary interfaces on demand and transform the program appropriately. This approach is taken by Caesar [19]. While this solution might me more convenient, it reliefs the developer from the burden of implementing the interfaces on its own, it is from a technical point-of-view no different.

A proxy can either be *open* or *closed*. A closed proxy, on the other side, is the only class which ever holds a reference to a component and no aliasing [14] of the component references takes place. In the following we use the term wrapper and proxy as a synonym for *closed proxy*.

Further proxies can either be replacing or forwarding. A replacing proxy processes a method call on its own and never relies on a component instance. A forwarding proxy executes functionality before or after forwarding the call to a component instance. We support both replacing and forwarding semantics. For this, the proxy class specified in the `extends` clause of the declare proxy statement must implement either a constructor with the following signature:

```
1   public MyProxy(Constructor c,Object [ ]args){...}
```

or with:

```
1   public MyProxy(T instance){...}
```

The constructor determines whether the proxy replaces (first case) or wraps the component instance (second case). In the first case the parameters of the component constructor are put into an object array and are passed to the proxy along with the original constructor. The constructor is necessary (e.g. for lazy initialization) if the object array alone would not allow the determination of the correct constructor (e.g. if a value is `null` the type can no longer be determined).

Additionally, a proxy must not only be able to wrap a component but also any other proxy generated for the same component. Otherwise, it would not be possible to chain proxies generated for a component by two or more declare proxy statements.

To sum up the generative semantics of our `declare proxy` construct (the properties are named **R1** - **R4**): generated proxies (**R1**) are closed, (**R2**) implement the interface of the component, (**R3**) must support the wrapping of other proxies as well as components and (**R4**) can be either replacing or forwarding.

2.3 Implementation Restrictions

Implementation restrictions are imposed on components annotated as `Wrappable` in order to ensure the following properties:

P1 it is possible to control the aliases of a component; that

[1] Thread synchronization is not shown in the listings because it is unrelated to the `declare proxy` statement.

the proxy is the only owner of a reference to a component.

P2 the use of proxies does not give rise to type errors at run time.

These properties depend on each other. We can control the references to a component if we are able to always use a reference to a proxy instance instead of a reference to a component instance. In this case, it is possible to transform the component such that it always returns its associated proxy instead of `this`. If now a newly created component instance is immediately wrapped by a proxy we can ensure that only the proxy holds a reference to the component. Note that we do not require any kind of alias protection mechanism [24, 8, 12, 7]. We only need to make sure that we are able to *confine* the reference to a component to a single proxy if needed; in other words, that we can control the aliases for components.

To guarantee the properties **P1** and **P2** the following restrictions need to be imposed:

- the type of the component must not to be used directly; neither in field-, local variable- or method declarations nor in type checks or type casts. ⇒ except for component creation, the only way to access a component is via its interface type(s).
- The supertype of a component must also be a component. Otherwise it would be possible to cast from a component interface type to the type of the superclass and invoke methods, which violates **P1**.
- Every subclass of a component is also considered a component and the first two restrictions apply.

To enforce these restrictions the IRC tool [10] can be used. It allows the definition of restrictions as checks on Java bytecode. We consider it important that these implementation restrictions are explicitly checked before the creation of proxies. Since all restrictions can be checked in one pass and no full program analysis is necessary the compile time overhead should be acceptable. Provided that a component complies with all implementation restrictions the generated proxies fulfill the requirements **R1-R4**

3. RELATED WORK

To the best of our knowledge, there has been no attempt to implement services such as passivation and pooling of components. So we can discuss only related implementation techniques.

The Dynamic Proxies [23] feature of Java can be used to create proxies. But, its usage would have the following drawbacks which makes it a "no-go": (1) **It leads to a tighter coupling between aspects.** The advantages of using proxies especially if they are used for optimizations vanish if each service that requires a proxy generates its own one. So, an aspect that generates a proxy and makes it available for other aspects would be required in order to enable the integration of multiple services using one proxy instance. In the proposed model this would happen implicitly if the `declare proxy` statements are identical. Further, the functionality to define precedence of aspects needs to be duplicated since the "normal" `declare precedence` functionality does not impose an order on the execution of `InvocationHandler`s (implementing the advice) associated with dynamic proxies. (2)**The generated dynamic proxies are open**. So, their use to simulate virtual references is very limited – as discussed especially in the motiviation. (3) **It forces to use reflection.** Even though the support for and performance of reflection is improving the source code remains harder to read and maintain and is also more error prone when compared with the proposed solution. This discussion applies equally well to AOP frameworkds based on the interceptor pattern [22] (e.g.: [3, 4, 21]) or native AOP languages (e.g.: [15, 19])

Hibernate [2] generates proxies at runtime using CGLib [1]. CGLib is more powerful than Java Dynamic Proxies – it provides default implementations for different types of proxies. Nevertheless, the base mechanism is comparable and thus suffers from the same problems as described above.

Generic Wrappers [5] is an language integrated approach to generate wrappers / proxies at runtime. E.g. the statement `class PassivationProxy wraps SessionComponent` generates wrappers for `SessionComponent`s and all subtypes of it. However, two main differences exist: (1) wrappers are not automatically used; to create a wrapper for a component it is necessary to explicitly instantiate a wrapper (e.g. `PassivationProxy p = new PassivationProxy<new` *AComponent()*`>()`) while in our case the proxies are automatically used. (2) The generated wrappers are subtypes of the wrapped objects - which means subclasses of components in our terminology. This is in our case not generally applicable – as discussed in the section on the design space of proxies.

Parametric Introductions [13] enable the parameterized introduction of e.g. methods and fields in existing classes. The introductions can be parameterized with the static information which will be available when the introduction is actually carried out, such as the type of the target class, the name of methods and their parameters, etc.. With a so-called unnamed introduction it would be possible to introduce complete sets of methods in a specific class without explicitly enumerating them. This mechanism could be used to introduce generic method implementations in proxy classes. However, as the name suggests parametric introductions can only be used to "introduce" code into an existing class or interface. The generation of completely new classes (the concrete proxy classes in our case) and the transformation of the components is out of scope for this approach. Nevertheless, it could supplement the generation of the proxy classes.

4. SUMMARY AND FUTURE WORK

In this position paper we have proposed an *inter-type declaration* to declare that specific classes are wrapped by automatically generated proxies. This gives us the ability to implement infrastructural services in a more straightforward manner than supported by current approaches. Further, the implementation is very robust since it only relies on types and not on names. The aspect's functionality is directly woven in the generated proxies by using standard pointcuts and advice. Further, it is shown which programming restrictions

needs to be imposed on the implementation of components and how to check them.

In future work we are going to investigate how proxies can help in the implementation of other infrastructural services. Additionally we are going to investigate if other mechanisms are needed to implement infrastructural services.

Acknowledgment

The author would like to thank the anonymous reviewers, Christoph Bockisch, Mira Mezini, Klaus Ostermann and Thorsten Schäfer for comments on earlier versions of this paper.

5. REFERENCES

[1] Code Generation Library (cglib). http://cglib.sourceforge.net/.

[2] Hibernate2 Reference Documentation. http://hibernate.bluemars.net/. Version 2.1.1.

[3] JBoss AOP. http://www.jboss.org/developers/projects /jboss/aop.

[4] Jonas Bonr and Alexandre Vasseur. Aspectwerkz. http://aspectwerkz.codehaus.org, 2003.

[5] Martin Büchi and Wolfgang Weck. Generic wrappers. In Elisa Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer Verlag.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – a System of Patterns*. John Wiley & Sons, 1996.

[7] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of OOPSLA 2003*, pages 374–387. ACM Press.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York. ACM Press.

[9] Linda G. DeMichiel. *Enterprise JavaBeans^{TM} Specification, Version 2.1*. Sun Mircosystems, 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 2003.

[10] Michael Eichberg, Mira Mezini, Thorsten Schäfer, Claus Beringer, and Karl Matthias Hamel. Enforcing system-wide properties. Melbourne, Australia. IEEE Computer Scociety. Proceedings of ASWEC 2004 (to appear).

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[12] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA 2001*, pages 241–255. ACM Press.

[13] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proc. of AOSD 2003*, pages 80–89. ACM Press.

[14] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest,Hungary. Springer.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of ECOOP 1997*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer.

[17] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, July 2002.

[18] R. Laddad. *AspectJ in Action*. Manning, 2003.

[19] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of AOSD 2003*, pages 90–99. ACM Press.

[20] OMG. *CORBA Components*. Object Management Group, June 2002. Version 3.0, formal/02-06-65.

[21] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Grard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of REFLECTION 2001*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan. Springer.

[22] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Software Design Patterns. John Wiley & Sons, 2000.

[23] Sun Microsystems. Dynamic Proxy Classes. java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[24] Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

[25] Markus Völter, Alexander Schmid, and Eberhard Wolff. *Server Component Patterns*. Software Design Patterns. John Wiley & Sons, 2002.