Integrating Orthogonal Middleware Functionality in Components Using Interceptors

Christoph Pohl and Steffen Göbel

Technische Universität Dresden
Institut für Systemarchitektur, Lehrstuhl Rechnernetze
D-01062 Dresden, Germany
pohl@rn.inf.tu-dresden.de, goebel@rn.inf.tu-dresden.de

Abstract Current component plattforms usually consider only a limited set of non-functional properties. Integration of these aspects is moreover handled in a rather static way. This article elaborates on possible uses of existing meta-programming facilities, notably interceptors, for custom integration of orthongonal middleware facilities. The concept is demonstrated with a concrete example of transparent client-side caching in Enterprise Java Beans. Further use cases exploiting the same principle in the contect of the COMQUAD project are presented in the second part.

1 Introduction

Todays middleware platforms like Enterprise Java Beans [5] follow a separation of concerns approach. Components are treated as black boxes, their business logic is opaque to the container's runtime environment. Middleware services are mostly integrated at deployment time, using declarative descriptors with meta-information about components' non-functional aspects. With common distributed component models like EJB, CORBA Components, Microsoft's COM+ and upcoming .NET [20], these aspects basically include transactional capabilities, synchronization, and security (authorization). Other aspects are not covered by their specifications and thus have to be integrated in vendor-specific ways. The authors already published on several of these non-functional aspects, like adaptation and caching [6, 18].

This article elaborates on how to seamlessly integrate these aspects with current component architectures, using *interceptors* as reflectional programming mechanism.

2 Related Work

Transparently adapting components' behavior is not a new idea: It actually dates back as early as 1982 when Smith published his thesis about reflection [19]. Although models have changed greatly since then, the basic concept is that components (programs) should have a notion about their current context and

(limited) control over their interpretative environment. This control is commonly referred to as *meta-programming*, i.e. programming at the abstract *meta-level* which is used to describe the executed code itself (classes, methods, etc.). *Meta-object protocols* (MOP) define interfaces to this meta-level.

According to [14], distributed systems are inherently predestined for reflectional programming, due to distribution transparency that is usually aimed at. Distribution itself can be viewed as a non-functional aspect that should ideally be separated from application logic by means of meta-programming mechanisms.

In distributed object-oriented systems, some sort of binding objects are typically employed as proxies to support location transparency. These proxies are often used as access points for meta-programming. For instance in the CORBA world, *smart proxies* have been developed as a simple meta-object protocol for altering behavior by intercepting calls from clients. Smart proxies provide better performance than *portable interceptors* as shown by [22], but they provide less flexibility and are not standardized by the Object Management Group[1]. Portable Interceptors have been in the focus of discussion for a few years and are now finally integrated in the OMG's CORBA standard [16]. Furthermore, smart proxy functionality can be implemented using interceptors[2] but not vice-versa.

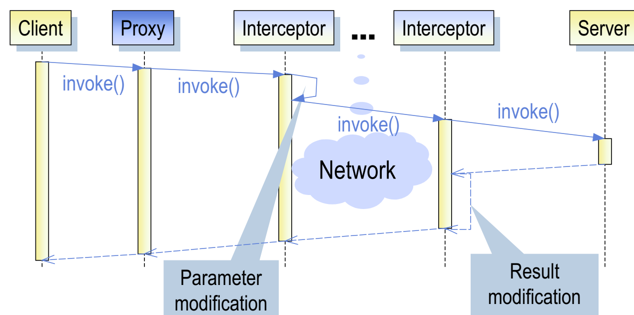

Figure 1. The principle of interceptors

The basic scheme of interceptors is shown in Fig. 1. On both client and server side, interceptors can be hooked into the control flow of (remote) operation calls, basically to add parameters and to augment results, but generally to alter virtually any property of a call's context, even its semantics. Interception may take place at different levels of call processing, hence there are request-, message-, object-, and network-level interceptors. The OMG's standard [16] currently defines only request- and object-level interceptors; most of the others will surely follow.

[1] Some vendors like TAO [23], Inprise, and Iona adopted and pushed the development of smart proxies but e.g. Iona is already discontinuing support for this feature.

[2] see [12]

Several recent publications [2, 21] leverage interceptors for building frameworks that more or less try to partly hide the complexity of meta-programming or to add a higher abstraction level. Our goal is rather to pinpoint possible uses of basic mechanisms that already exist in current component platforms.

3 Approach

This section first presents how non-functional services have been integrated in available middleware using interceptors, instancing caching as a practical example. The second part elaborates on future prospects for further integrations in the field of adaptation.

3.1 Caching

While many existing distributed component-based applications follow a clear thin-client approach, fat clients still are eligible in certain scenarios with high user interactivity, e.g. e-Learning and collaborative multimedia. In these cases, the common issue “Either data must come to the process or the process must come to the data” can hardly be decided in the latter sense because this would require network round-trips upon every interaction. On the other hand, efficient client-side data access requires partial replication, which eliminates unnecessary network calls for already queried component attributes, to provide the appropriate locality of reference.

The authors already showed in previous publications [18] how caching functionality, a special form of partial replication, can be integrated using smart-proxy-like meta-programming techniques. This involved augmenting default (Java RMI) proxies. Unfortunately, deployment of modified component stubs proofed to be too complicated on most existing container (EJB) platforms because of the numerous providers' varying architectural concepts. For this reason, the current prototype is now based on an open-source EJB container's built-in facilities for reflectional programming.

Integration with interceptors in JBoss. JBoss [10] was chosen for being a vivid project with numerous cutting-edge software-architectural features, including a framework for request-level interceptors as introduced in Sec. 2. Client-side integration of these interceptors is shown in Fig. 2: Component proxies are transparently generated and instantiated using Java's Dynamic Reflection API. A `ClientContainer` passes each request through a chain of `Interceptor`s, whose sequential order is determined by the bean provider or application assembler at deployment time. The last interceptor always hands the request to the appropriate `InvokerProxy` – JRMP in Fig. 2 for Java Remote Method Protocol – that finally calls the server. Server-side interceptors are stacked in a similar fashion. When a response returns from the server, it passes through the same interceptor chain in reverse order as already shown schematically in Fig. 1.

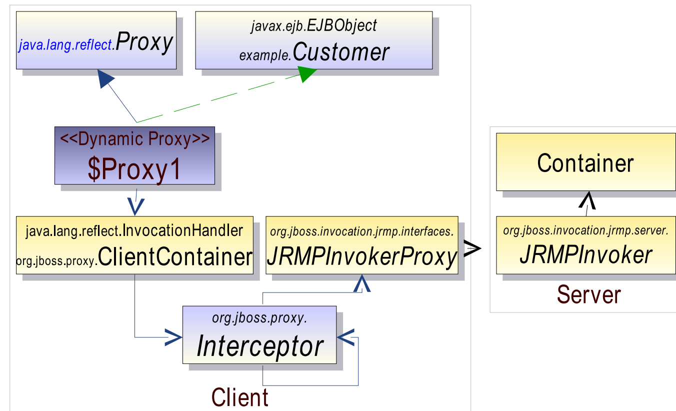

Figure 2. Interceptors in JBoss

Even though CORBA Portable Interceptors are already part of Sun's Java 2 Standard Edition 1.4 and JBoss's IIOP provider JacORB [7], JBoss does not yet employ the CORBA API for integrating interceptors because it doesn't use IIOP but JRMP by default for performance reasons. Fortunately, `org.jboss.proxy.Interceptor`'s functionality can be viewed as a sub-set of `org.omg.PortableInterceptor.Interceptor` which should make future changes easy.

The integration of transparent client-side caching of component attributes basically relies on adding a `CachingInterceptor` to a component's `jboss-container` descriptor during deployment as the second link in the client interceptor chain between `Home-` / `EntityInterceptor` and `SecurityInterceptor`. JBoss's interceptor API basically contains a method `Object invoke(Invocation i) throws Throwable;` where i contains all necessary information about the ongoing request, especially the component's identity, called method and parameters. Assuming, that component attributes are typically exposed at an EJB's remote interface via Java-Beans-style `getXyz()` methods, this information is sufficient for attribute caching: The `CachingInterceptor` checks all `Invocations` i against its back-end and returns the request's response immediately if the result can be served by the local cache. Otherwise, returning responses from not yet cached invocations are automatically filed in the cache.

Cache back-end and multiple reference handling. Prototypically, the cache back-end was implemented using a simple `Hashtable` with component identity, method and parameters as combined keys and results as values, i.e. $(i, m, \{p\}) \rightarrow r$. A more sophisticated version uses a *JCache*[3]-like API that caters for memory management, persistent caching etc. The basic granularity of

[3] Submitted Java Specification Request [3], similar to Oracle's *Object Caching Service for Java* (OCS4J).

cached data is per-attribute but these are members of identifiable components which enables collective invalidation of attributes.

As already elaborated in [18], *multiple reference handling* is also an important issue for caching services in distributed component middleware. Component references are typically passed around as marshaled objects (proxies / stubs) which makes it possible for a client to obtain a number of proxy objects for one and the same remote entity. This is counterproductive for memory consumption. `CachingInterceptor`s have also be leveraged to support efficient multiple reference handling by checking all returned remote references, i.e. proxies, for duplicates in the local cache. As Sun's EJB specification [5] explicitly discourages direct equality checking between entities using `equals()`, and `isIdentical()` may result in additional undesired remote calls, EJB handles[4] are held liable for component equality which is necessary for duplicate checking. Note that special attention must be paid when collections of proxies are returned, e.g. when accessing $1 : n$ relationships between entities. Corresponding accessor methods have to be tagged (see below) before generating caching code which ensures that every member of returned collections is checked for duplicates in the local cache.

UML and Computer-Aided Software Engineering. Simply caching every component attribute the same way is quite naive; some means of configurability would be appreciated. This can be accomplished by defining stereotypes according to the Unified Modeling Language (UML) [17] for tagging component attributes as shown in Fig. 3.

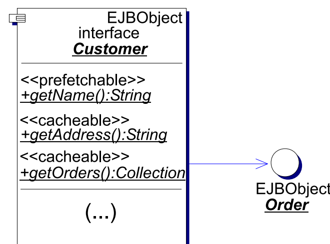

Figure 3. UML stereotypes for component attributes

Non-tagged attributes are considered *volatile* by default, i.e. their values are considered to be subject of frequent changes which makes them inappropriate for caching. The stereotype *cachable* denotes all attributes that usually change irregularly and *prefetchable* are frequently used attributes that should be transfered immediately. This is especially useful in combination with *Value Objects*[5].

[4] JBoss's proxy implementations already transfer handles upon initialization, hence no additional network round-trips are needed.

[5] Introduced in [9], this pattern became quite common for reducing network round-trips by bulk state transfer.

The counterparts of stereotypes in Java source code are JavaDoc tags – special commands or attributes embedded in code comments which where originally intended for formatting purposes of API documentations. So, a stereotype << *cacheable* >> becomes a `/** @cacheable */` comment above an accessor method. The auspicious code generation utility *XDoclet* [15] is used to evaluate these tags and create additional classes, adapt deployment descriptors etc. XDoclet follows the *one source approach*, i.e. one source file contains all necessary information about all supplementary classes, utilities, descriptors. It provides merge points in its generator templates for adding specific parts in descriptors or even for integrating custom templates capable of creating auxiliary classes.

Deployment. JBoss provides flexible ways for integrating custom deployers, i.e. server components capable of processing certain types of component archives, depending on file names, contained descriptors etc. This necessitates only the implementation of a `CachingDeployer` for scanning component archives comprising special caching descriptors. This deployer is integrated using JBoss's JCA[6] implementation and referenced by name from deployed components. Thus component properties concerning cachability of attributes are dynamically installed in a client's runtime environment with the component's classes.

Synchronization and Consistency. Update synchronization is another important challenge for distributed caching systems. Most publications define graded consistency levels [11, 4] because guaranteed transactional consistency would require two phase locking (2PC) which would in turn entail additional network round-trips that have been economized before with the introduction of caching.

The simplest solution are regular cache purges, automated by a background thread in the caching back-end that deletes cached objects after their expiration limit has been reached . Expiration limits are configurable via source code tags in the above described manner. Unfortunately, the *lag* until a given attribute update reaches the last client equals the defined expiration time in the worst case. This implies that expiration times have to be adapted to an application's least tolerable lag. On the other hand, much of caching's savings get spoiled if expiration times are much smaller than typical update cycles. Only heuristics lead to appropriate adjustments.

Consequentially, the authors currently experiment with more sophisticated update propagation algorithms. Instead of mixing the caching back-end's memory management strategy, e.g. least recently used (LRU), with consistency issues like expiration times, the caching interceptor should take care of these problems itself. Some means of multicast or publish-subscribe propagation seem to be at hand, where either modifying clients themselves notify other replicates (*write-through*) or let a component's originating server do so (*write-back*) [1]. Existing proposals like [11, 8, 13] are either intended for use in intranets or they simply abstract from underlying communication protocols. But todays typical network

[6] *J2EE Connector Architecture*, see http://java.sun.com/j2ee/connector/.

scenarios feature network address translation (NAT) devices, firewall configurations and similar obstacles that make direct server-to-client connections, i.e. clients logically become servers, difficult if not impossible. For this reason, protocols should preferably be client-initiated.

A possible implementation of such a client-initiated protocol could regularly poll its leased component's servers with a list of component identities which would be replied with a list of update time stamps of these components which can be drawn on for cache invalidation. This performs poorly for large numbers of component references per client. So, a slightly better solution is to let clients send a list of name-value-pairs $(identity, timestamp)$ to which servers reply only with a list of updated identities, as they also know the current timestamps for all identities. Directly replying with the updated objects instead of their identities proofed to be less suitable. Anyway, a fast lookup facility for update timestamps is required for all of these solutions, e.g. a stateless session bean.

Updates can also be transfered selectively without having to initiate direct server-to-client connections by using either a JMS[7]-topic with clients as subscribers or a piggy-back approach as it is common for TCP acknowledgements. An additional server-side interceptor notices which component references are handed out to which clients, enabling it to keep track of necessary notifications. This solution is optimal in respect to bandwidth efficiency but it scales poorly on server side because of the tremendous overhead for state management of client-component-timestamp-relations.

Relations between components have already been mentioned above in the context of multiple reference handling but they also become important in the context of synchronization and consistency: Speaking in terms of UML [17], these relations can be categorized into *associations*, *aggregations* and *compositions*. Especially the latter ones imply life-cycle-dependencies of their members, meaning that certain operations on one member affect the validity of the other member's cache. Up to now, such dependencies had to be analyzed manually, corresponding operations had to be tagged by hand. Ways of automating this tedious task are currently being worked on.

3.2 Adaptation

This section is to illustrate the use of interceptors in the context of the *COMQUAD*[8] project. Component-based software engineering has become more and more important in the last few years. However, two important properties of modern distributed systems are not yet fully considered in existing concepts of component based systems:

---

[7] *Java Message Service API*, see http://java.sun.com/products/jms/. Although JBoss's own implementation JBossMQ (formerly spyderMQ) is not yet firewall-enabled, there already exist commercial tunneling solutions for JMS.

[8] *COM*ponents with *QU*antitative properties and *AD*aptivity. DFG-funded research group. See [6].

- Quantitative properties such as network bandwidth or response time can neither be specified nor guaranteed in existing component models.
- Dynamic component adaptation is not explicitly supported. Components are rather relatively static constructs that are difficult to adapt to changing system and user environment. A possible classification of adaptational issues is shown in Tab. 1

Classification	*Adaptation*	*Description*	
Who?	User-driven	administrator	
(Trigger)	System-driven	container	
Where?	Parametric	within component	
	Structural	within application	
What for?	System-side		
	User-/application-side		
When?	Static	deploy-time	
	Dynamic	run-time	upon initialization
			during connection
How much?	(Overhead)	additionally needed time & resources	

Table 1. Classification of adaptation

The interplay of these two properties on changing quantitative properties of an application is especially important. The COMQUAD project aims at the development of a system architecture in conjunction with a development methodology supporting the composition of adaptable software from components with consideration of non-functional properties. Examples of non-functional properties are bandwidth, processing time, throughput, delay, jitter, and also security properties.

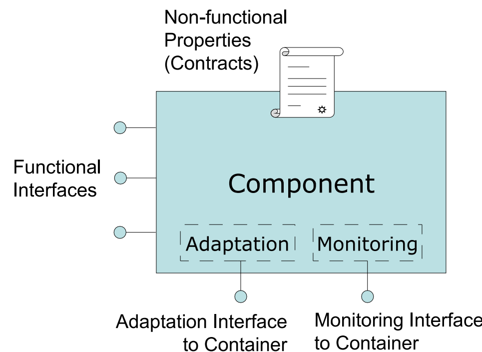

Figure 4. COMQUAD component model

In order to achieve these aims the classic component model has been enhanced, consisting of interfaces and binary code with monitor and adaptation manager (cf. Fig. 4). The adaptation manager contains the logic for the dynamic reconfiguration of the component. It can be generated from a declarative description of the adaptation behavior or manually programmed by developers. The monitoring part of the component is necessary to observe the assured contracts at run-time. Meta-programming facilities like interceptors offer a conveniently simple way to obtain this information and make it available to the components. A component's container can transparently add appropriate interceptors at deploy-time or even upon initialization. The results of the monitoring are forwarded to the adaptation manager by the container in order to trigger the adaptation processes.

Monitoring. In the following we want to investigate which information can be gained by means of interceptors:

Invocation statistics. Statistical information about invocations includes frequencies of method calls and number of clients. These data are valid in a particular interval and they are required to to monitor periodical tasks and to detect potential server overloads. The statistic belongs to component instances, component classes or the entire server. An monitoring interceptor has to maintain a table for all component instances, component classes, and the entire server.

Data throughput. Data throughput is the amount of data transmitted by method invocations from client to server and back in a given interval. By means of the Java serialization mechanism an interceptor is able obtain this value both at the client and the server. Furthermore, the data throughput can be divided into data of component instances, component classes, and the entire server.

Processing, response and delay times. The response time of a remote method invocation consists of the processing time of the method at the remote server, network transmission time, and the time for marshalling/unmarshalling parameters at client and server. These parameters are potentially very important in real-time applications. The processing time is measured by means of a server-side interceptor as the interval between forwarding a method invocation to a component instance and its completion. In a similar way, the response time is measured by a client-side interceptor.

The delay time of a remote method invocation is the interval between method calling by the client and starting method processing by the server. In contrast to response and processing time, this value is difficult to measure in a distributed environment without exactly synchronized clocks. At least it is possible to get good approximation. The difference between response time and processing time results in the time for sending and receiving method invocations over the network. If a constant network bandwidth is assumed the ratio between sending time (another name for the delay time) and data size of sent parameters is the

same as the ratio between the receiving time (transmission from server to client) and data size of return parameters. Hence, the delay time can be calculated.

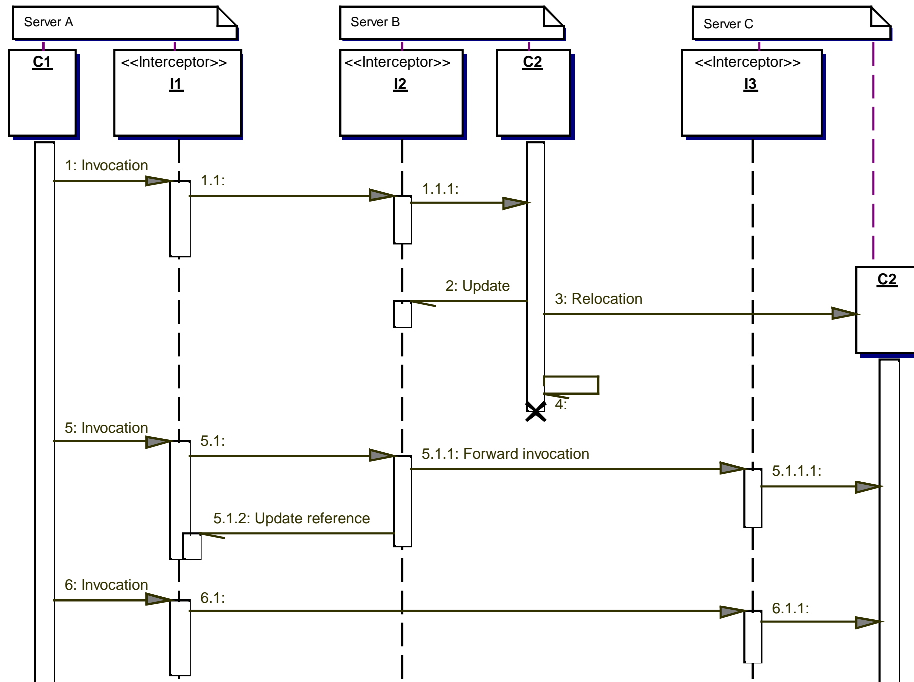

Figure 5. Transparent relocation of components with interceptors

Relocation of components. Interceptors can also be used to achieve transparent relocation of components at run-time which is an important adaptation mechanism. Components usually obtain references to other components by means of a name service (JNDI in case of EJB) at run-time and then store these references in a local variable for later usage. If a component is relocated to another server, all references from other components become invalid. To avoid this, an interceptor could forward a particular method invocation to the new location of the component, because all method invocations must pass an interceptor. Hence, all component relocations must be reported to the interceptor that maintains a table of old and new component references.

A sequence diagram of a possible relocation scenario is depicted in Fig. 5. After relocation of component $C2$ from server B to server C any method call to this component using an old reference is forwarded to the new location by interceptor $I2$. Obviously, this leads to a higher delay and response time for the client due to the additional step in the method call, but only for the first method

call. Later on, the client interceptor of computer A redirects any further method invocations to $C2$ to the new location at server C. Prior to this, interceptor $I2$ sends a location update about component $C2$ to interceptor $I1$. The advantage of this solution is that every relocation of a components only requires one additional redirection. The processing time of the two interceptors at client and server can be disregarded in comparison to the network transmission time. As a drawback, local communication between components within one server must also pass through interceptors.

4 Conclusion

This article presented how non-functional aspects can be easily integrated in todays middleware plattforms using existing meta-programming facilities, notably interceptors. Linking-up with software design and modeling facilities was demonstrated exemplary, as well as leveraging of code generation tools. An outlook was given to open issues the authors will tackle next, i.e. finding optimal cache synchronization strategies for given usage scenarios, automated component dependency analysis, and reference faulting mechanisms for more flexible cache memory management.

The second part elaborated on possible uses advanced component concepts in the *COMQUAD* project [6]. Especially for adaptational purposes, interceptors present appropriate ties for monitoring and relocation of components. Other synergies are currently being evaluated.

References

[1] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Practice and Experience*, 4(5):337–355, August 1992.

[2] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. IFIP, Springer-Verlag.

[3] Jerry Bortvedt. JCache - Java Temporary Caching API. Java Specification Request #107, 19 March 2001.

[4] Gregory Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a caching service for distributed CORBA objects. In *Middleware'00*, pages 1–23, 2000.

[5] Linda G. DeMichiel, L. Ümit Yalčinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification Version 2.0.* Sun Microsystems, final release edition, 14 August 2001.

[6] Technische Universität Dresden. COMponents with QUAntitative properties and ADaptivity (COMQUAD). Project homepage: http:// www.comquad.org/, August 2001. DFG research group.

[7] Freie Universität Berlin and Xtradyne Technologies AG. JacORB. Project homepage: http:// www.jacorb.org/.

[8] Arun Iyengar. Design and performance of a general-purpose software cache. In *Proceedings of the 18th IEEE International Performance Conference (IPCCC'99)*, 1999.

[9] Design patterns catalog. In *J2EE Design Patterns*. Sun Microsystems, 2001.

[10] JBoss Group. JBoss. Project homepage: http:// www.jboss.org/.

[11] Rammohan Kordale, Mustaque Ahamad, and Murthy V. Devarakonda. Object caching in a CORBA compliant system. *Computing Systems*, 9(4):377–404, 1996.

[12] Rainer Koster and Thorsten Kramp. Loadable smart proxies and native code-shipping for CORBA. In *Proceeding of the 3rd International IFIP/GI Working Conference, USM*, volume 1890, pages 202–213. Springer, 2000.

[13] Vijaykumar Krishnaswamy, Ivan B. Ganev, Jaideep M. Dharap, and Mustaque Ahamad. Distributed object implementations for interactive applications. In *Middleware 2000*, 2000.

[14] Jeff McAffer. Meta-level architecture support for distributed objects. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, pages 39–62, 1996. Published before in IWOODS'95.

[15] Rickard Öberg, Andreas Schaefer, Ara Abrahamian, Aslak Hellesøy, Dmitri Colebatch, and Vincent Harcq. XDoclet. Project homepage: http:// xdoclet.sourceforge.net/.

[16] Object Management Group. *CORBA Portable Interceptor Specification*, March 2001. ptc/01-03-04, formal/02-05-18.

[17] Object Management Group. *Unified Modeling Language, v1.4*, September 2001. formal/01-09-67.

[18] Christoph Pohl and Alexander Schill. Middleware support for transparent client-side caching. In Elke Pulvermüller, Isabelle Borne, Noury Bouraqadi, Pierre Cointe, and Uwe Assmann, editors, *European conference on Theory And Practice of Software ETAPS'02*, volume 65 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers. Software Composition Workshop.

[19] Brain Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., February 1982.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, November 1997.

[21] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *International Conference on Software Engineering*, pages 233–242. IEEE, 2001.

[22] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt, and Ossama Othman. The design and performance of meta-programming mechanisms for object request broker middleware. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[23] Washington University, St. Louis and University of California, Irvine. Real-time CORBA with TAO. Project homepage: http:// www.cs.wustl.edu/ ˜schmidt/ TAO.html.