Aspects in Distributed Environments

E. Pulvermüller, H. Klaeren, and A. Speck

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
D-72076 Tübingen

Abstract. We illustrate how to combine CORBA as a distributed system with aspect-oriented programming (AOP) and the resulting positive impacts. In particular, we focus on the question how AOP can be applied to a CORBA application aiming at a better separation of concerns.
The paper shows with implementation examples a practical way to package the distribution issues of a CORBA application into separated aspects (with AspectJ). This supports the distributed application design and leads to increased flexiblity.

1 Introduction

A system with distribution is characterized by a substantial amount of complexity. When designing distributed applications additional issues compared to centralized systems have to be considered. Some of these issues are fragmentation, replication, naming, concurrency, failure, configuration and communication [25].

There are different approaches to manage this inherent complexity in distributed systems:

One way is to aim at *transparency* to build models which are very close to those of centralized systems. Distribution related issues are hidden to the user and the application programmer [3]. However, there are many cases where full transparency is not desirable or reachable. In [9] it is even pointed out that distribution transparency is impossible to achieve in practice.

In a fully transparent communication environment the application developer has at best limited possibilities to influence the way of communicating or to improve network performance, respectively. The flexibility to fully exploit communication knowledge is restricted.

Other approaches aim at supporting the development of distributed applications by offering a *process* or *method*, respectively. In [25] a stepwise development process is shown. The different concerns (fragmentation, replication, naming, concurrency, failure, configuration, communication) are handled transparently according to the idea of separation of concerns.

Besides transparency and the application of an appropriate software development method, the usage of *patterns* [7], [2], [19], particularly those for distributed systems as in [16] is an established means to ease and improve application design [22]. Patterns provide architectural or design solutions for known problems.

Distributed application development can furthermore be simplified using a *framework*. Examples for such frameworks are CORBA implementations [18], [24] and ACE [23]. A framework is a semi-finished set of cooperating classes which are the main components to create systems [6], [19], [20]. A developer customizes the framework to a particular application.

Leaving fully transparent systems out of account due to their lack of flexibility, there still remains in all approaches the necessity to inject additional code into objects. Distribution issues like communication, naming, etc. are implemented this way. This is one primary source of complexity in distributed applications which can be addressed with aspect-oriented programming (AOP) [13], [14]. AOP is a concept to achieve separation of concerns. Reusable objects are designed and coded separately to cross-cutting code. An aspect weaver is responsible for merging both design elements. AOP is one research area in the field of generative programming [5], [4].

In the remainder of this paper we show how a CORBA environment can be profitably combined with aspect-oriented programming. We take JavaIDL (together with the IDL compiler `idltojava`) [26] as an almost CORBA 2.0 [18] compliant distributed object technology provided with the Java Development Kit (JDK) 1.2. For the AOP part we use AspectJ 0.3beta1 [28]. This is an aspect-oriented extension to Java developed at Xerox PARC. AspectJ 0.3beta1 supports additional syntax to define aspects, separated from the Java class code. Currently, the weaving output is pure Java which can be automatically compiled with JDK.

2 Combining CORBA and AOP

When combining two technologies the question raises how one technology can take advantage of the other. With AOP and CORBA systems there are multiple answers to this question:

1. Applying aspects to cross-cutting issues of the problem domain
   Aspect-oriented programming can be applied to application objects running in a CORBA environment in the same way as in centralized applications. Cross-cutting code resulting out of the problem domain is collected in aspects (cf. figure 1)[1]. After the weaving process (e.g. with AspectJ) the resulting pure Java application can run distributed in a CORBA environment that supports the Java language mapping (e.g. JavaIDL from Sun Microsystems or VisiBroker from Inprise).
2. Implementing CORBA related code in distributed applications as aspects
   Regarding a CORBA distributed application, both client and server are supplemented with more or less code referring to ORB and Common Object

[1] Weaving should be regarded just as one phase of the compilation process [12]. Throughout this paper we present the weaving as a separate primary phase in order to stress it.

Services (defined in COSS [17]) or even Common Facility (CF) functionality. Program parts concerning distribution issues could be separated from the original objects by means of aspects (cf. figure 1).

3. Distributing aspects using CORBA
   Many objects can reuse the same aspect. In a distributed environment this means that these objects could be distributed. Therefore, the reused aspects have to be made available in the whole distributed environment. This increases complexity since the suitable aspect has to be found in a network of computing units. The emerging complexity caused by the need to manage those distributed aspects could be handled by a kind of aspect repository similar to an Interface Repository. This could help managing those distributed aspects. Especially graphical tools could ease locating the aspects in the network.
   In figure 2 an *Application* at location C can find the appropriate aspects via the aspect repository (e.g. *Aspect1* and *Aspect2*). With dynamic aspects as proposed in [15], [11] the process of searching and choosing aspects can be done even at run-time.
4. Using aspects to implement or extend a CORBA (like) system
   In [10] a CORBA-compliant middleware architecture for distributed systems is described. The idea of this adaptable and configurable middleware, called AspectIX, is to extend CORBA to a more open and flexible model. In AspectIX a client may configure (even dynamically) functional and nonfunctional properties of the objects semantics by means of aspects.

Section 3 concentrates mainly on the question in which ways CORBA related code can be put into aspects (item 2). Since distribution related code has to be injected into object code this is a major source of complexity. Therefore, applying AOP to distributed CORBA applications this way addresses one of the most urgent problems.

3 Aspects for CORBA Communication Code

In this section we want to work out in which ways aspects could be applied to applications using a CORBA-compliant ORB implementation for communication purposes. The question is whether AOP can help to improve problem understanding, coding and reuse. The CORBA program parts have to be investigated in order to find the possible positive impacts which aspects have on the resulting CORBA code.

In the following a simple client/server application exchanging a string is investigated[2]. With the simple example the impacts of AOP can be shown.

The implementation uses JavaIDL as an almost CORBA-compliant implementation from Sun Microsystems and AspectJ 0.3beta1 from Xerox as an AOP mechanism.

[2] Both error handling and debugging functionality are classical examples for the usage of aspects [28]. They are not regarded in this paper.

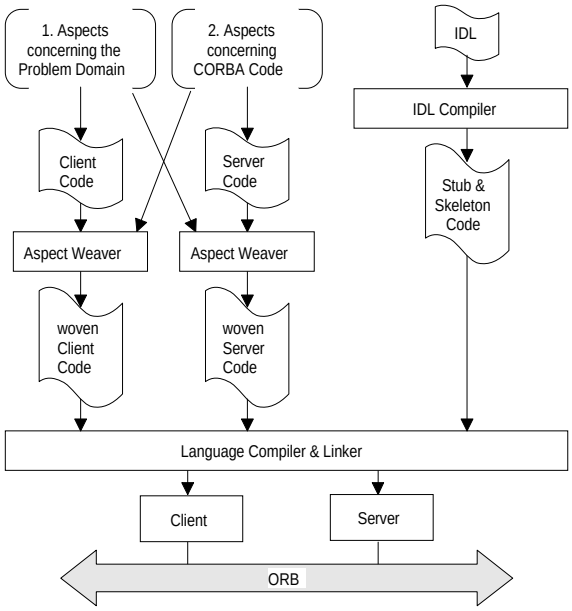


Fig. 1. Cross-cutting Code relating to (*1.*) Problem Domain and (*2.*) CORBA as Aspects

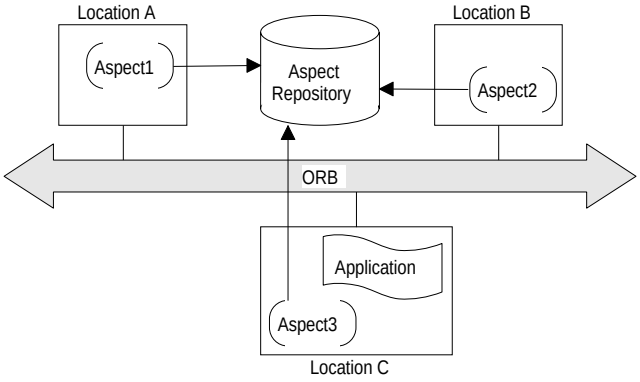


Fig. 2. Distributed Aspects

3.1 The Client Side

The client can be coded almost as if it were a pure local client.

```
import App.*;

public class Client
{
  static Interface_Name ref;

  public static void main(String args[])
  {
    try{

      // Call server object and print result
      String aString = ref.get();
      System.out.println(aString);

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
      }
  }
}
```

The variable `ref` holds a reference to the `Servant`. The servant implements an interface called `Interface_Name` which defines the `get()` operation. In the distributed version the interface is generated out of the `.idl` file. `App` is a package containing the stubs generated with the IDL compiler `idltojava` or other application specific packages in a local version.

Additional code for a local version is encapsulated in the following aspect:

```
/**
  * The ClientLocal aspect injects code before method main
  * of class Client to run the Client locally.
  */
aspect ClientLocal {
  advise void Client.main(String args[]){
    static before {
      try{
         ref = new Servant();
      } catch(Exception e) {
           System.out.println("ERROR : " + e);
           e.printStackTrace(System.out);
        }
    }
  }
}
```

For the interaction with a server in the CORBA environment the communication and the name server issues are localized in a client communication aspect. It is assumed that the corresponding server works with the COS naming service [17] of JavaIDL.

```
/**
  * The ClientComm aspect injects CORBA code before
  * method main of class Client.
  */
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

aspect ClientComm {
```

```
  advise void Client.main(String args[]){
    static before {
      try{
        // Create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // Get the root naming context
        org.omg.CORBA.Object objRef
          = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // Resolve the object reference
        NameComponent nc = new NameComponent("Name", " ");
        NameComponent path[] = {nc};
        ref = Server_InterfaceHelper.narrow(ncRef.resolve(path));
      } catch(Exception e) {
          System.out.println("ERROR : " + e);
          e.printStackTrace(System.out);
        }
    }
  }
}
```

Now, the corresponding server may run locally or in the CORBA distributed environment. For a locally running server (class `Servant`) the client has to be woven with the `ClientLocal` aspect whereas a distributed server leads to the need to weave with the `ClientComm` aspect.

3.2 The Server Side

Further advantages of AOP can be exploited if it is applied to the server code. Whereas distribution issues were separated on the client side, the main advantage considered on the server side is flexiblity. With AOP, the application developer can combine the appropriate aspects into the server compilation process, which results in a server with a special functionality.

In our example we demonstrate how a server can provide either a name server or a file based (stringified object reference [18] stored in a file) functionality depending on the aspects chosen in the compilation process. Obviously, a client using this functionality has to implement the corresponding code (at best encapsulated in an aspect, as well).

The application can be divided into kernel code and some flexible, interchangeable code fragments packed into aspects as depicted in figure 3.

The `Server` is responsible for creating a `Servant` instance which serves the request from the client.

```
import App.*;
import org.omg.CORBA.*; // CORBA specific classes.

class Servant extends _Interface_NameImplBase
{
  public String get()
  {
    return "\n A String \n";
  }
}

public class Server
```

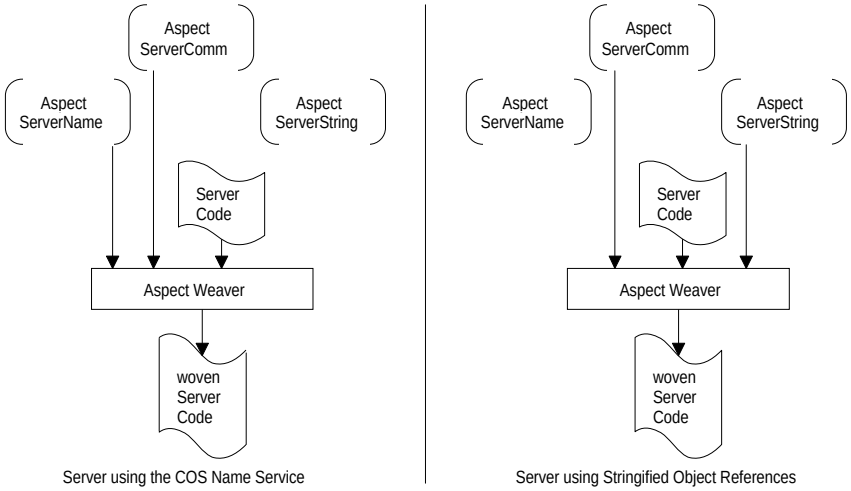


Fig. 3. Flexible Programming with Aspects

```
{
  static Servant ref;

  public static void main(String args[])
  {
    try{
      // Create the servant
      ref = new Servant();
    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
  }
}
```

As shown in figure 3 there is an aspect implementing the CORBA related code using the name server functionality:

```
/**
  * The ServerName aspect injects name server
  * functionality after method main of class Server.
  */
import org.omg.CosNaming.*; // naming service.
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

aspect ServerName {
  advise void Server.main(String args[]) {
    static after {
      try{
        // Create and initialize the ORB
        ORB orb = ORB.init(args, null);
        orb.connect(ref);

        // Get the root naming context
        org.omg.CORBA.Object objRef
                = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

```
        // Bind the object reference in naming
        NameComponent nc = new NameComponent("Name", " ");
        NameComponent path[] = {nc};
        ncRef.rebind(path, ref);
      } catch(Exception e) {
          System.out.println("ERROR : " + e);
          e.printStackTrace(System.out);
        }
    }
  }
}
```

The `ServerString` aspect implements the CORBA related code, as well. Instead of using the name server it saves a stringified reference into a file out of which the client can extract the reference.

```
/**
  * The ServerString aspect injects stringified reference
  * code after method main of class Server.
  */

import java.io.*;
import org.omg.CORBA.*;

aspect ServerString{
  advise void Server.main(String args[]) {
    static after {
      try{
        // Create and initialize the ORB
        ORB orb = ORB.init(args, null);
        orb.connect(ref);

        // stringify the ref and dump it in a file
        String str = orb.object_to_string(ref);
        String filename = System.getProperty("user.home")
           +System.getProperty("file.separator")+"IOR";
        FileOutputStream fos
           = new FileOutputStream(filename);
        PrintStream ps = new PrintStream(fos);
        ps.print(str);
        ps.close();
      } catch (Exception e) {
          System.err.println("ERROR: " + e);
          e.printStackTrace(System.out);
        }
    }
  }
}
```

The `ServerComm` aspect can be reused. It contains the code responsible for waiting for a client request. This implementation is used both in a server working with the name server and in the server using the stringified reference approach.

```
/**
  * The ServerComm aspect injects the wait for request
  * code after method main of class Server.
  * (must be inserted after ServerString or ServerName).
  */
import org.omg.CORBA.*;

aspect ServerComm {
  advise void Server.main(String args[]){
    static after {
      try{
```

```
        // Wait for invocations from clients
        java.lang.Object sync = new java.lang.Object();
        synchronized(sync){
          sync.wait();
        }
      } catch(Exception e) {
          System.out.println("ERROR : " + e);
          e.printStackTrace(System.out);
        }
    }
  }
}
```

These examples show that aspects are a promising and powerful means to achieve flexible programming by choosing and combining the necessary aspects.

Furthermore, an aspect can implement a non CORBA-compliant functionality. When exchanging this aspect with a CORBA-compliant one an application easily can be ported to a CORBA system. Besides this exchange of aspects no further changes may be necessary if the CORBA related code is completely concentrated into separated aspects.

There are other approaches aiming at a good separation of concerns. One way would be to use abstraction, i.e. to encapsulate CORBA related code in functions. This requires to insert an appropriate function call into the client or server code. Unlike the aspect solution the developer is obliged to change this call within the code when functionality has to be changed. With aspects, the developer has a clearly separated and therefore flexible programming construct. Moreover, a solution with aspects is orthogonal to the abstraction provided by the functions.

4 Experiences with AspectJ in a CORBA Environment

JavaIDL seems to be a very promising and almost CORBA-compliant ORB since it is integrated into JDK 1.2. According to Sun JavaIDL supports the basic CORBA-compliant ORB interfaces as well as a COS Naming Service [26]. However, currently not all CORBA features are implemented in JavaIDL[3] and it is available on a very limited number of platforms (Solaris and Windows NT). It may be expected that the significance of JavaIDL will increase due to the fact that Sun Microsystems has announced that it will support the porting of JDK 1.2 to other platforms (e.g. Linux).

Though all the distributed applications using JavaIDL have been tested on Windows NT 4.0, weaving with AspectJ 0.3beta1 is also possible on Linux if appropriate JavaIDL packages exist.

Despite promising advantages there remain some minor problems:
Multiple existing aspects can be woven with application objects. This requires the possibility to specify the order of the aspects that are woven into the application code. The weaving priority in AspectJ 0.3beta1 is determined by the aspect name. A language construct to express this order explicitly would ease the aspect management.

[3] The Interface Repository, for example, is not yet implemented.

In the case of an error during compilation, the error report of the compiler refers to the woven file. The error information currently has to be manually mapped back into the source code.

Our examples have shown how CORBA functionality can be extracted into aspects. This concept of separating CORBA communication issues from the application objects by using AOP leads to clear structured code. The programmer can concentrate on the implementation of problem domain issues.

Furthermore, functionality concerning distribution can easily be reused if placed into aspects (this was shown in section 3.2 with aspect `ServerComm`).

5 Conclusion

Our goal was to examine how AOP can be combined with CORBA. The work focused on the question whether and how aspect-oriented programming can improve distributed application development in a CORBA environment. It has led to the result that packaging the distribution issues of a CORBA application into separated aspects can support the application developer.

Aspect-oriented programming with AspectJ applied to a CORBA system proved to be a helpful means to achieve both a kind of transparency by separation of concerns and flexibility for the application programmer. CORBA related code can be extracted and encapsulated in aspects. When implementing the objects concerning the problem domain the programmer is not confused with distribution issues. Despite encapsulating distribution issues flexibility is preserved. The application programmer can fully decide about CORBA related issues when implementing the aspects. Furthermore, available aspects containing CORBA related code may be exchanged easily.

Further work should be done to find a general way how all CORBA ORB and COS related code could be placed into aspects. This could ease and improve CORBA application development. Furthermore, the application of AOP to CORBA system development (which is only touched in this paper) should be investigated in detail. Especially the problem of semantically interacting CORBA services may be addressed by AOP to support the CORBA system designer.

In this paper we concentrated on AOP via the application of AspectJ. A variety of other AOP mechanisms exist (e.g. Composition Filters [27]). Each of them may lead to improvements if used for CORBA applications or systems.

Moreover, a network wide (re)use of aspects should be supported. This may be realized by additional tools similar to an Interface Repository for interfaces in CORBA. Graphical tools could be helpful.

Dynamic aspects as proposed in [15], [11] could improve flexibility. Therefore, aspects implementing specific CORBA functionality could be exchanged not only at compile-time.

The examples in this paper show that a mechanism assuring the correct combination of aspects is necessary. In section 3.2 we introduced the aspects `ServerString` and `ServerName` for the remote communication between client and server. We have to assure that only one of both aspects is chosen. Moreover,

weaving the `ServerName` aspect requires that `ServerComm` is woven as well and that the client is woven with an aspect containing the name server issues instead of the stringified issues. Composition validation algorithms as proposed in [1] for the composition of components may be useful to address this problem. The concept of assertions [8] may contribute to a solution, as well.

Another open question is how to distribute objects. Effective distribution strategies are needed to achieve goals like maximizing performance and minimizing costs. There exist only a few guidelines for effective object distribution [21]. When distributing aspects, a similar problem occurs. Further research in this field is necessary.

Guidelines for good aspect design would be extremely helpful. Such guidelines may be embedded into a software development method which considers the usage of aspects.

Acknowledgments. We would like to thank Prof. Eisenecker and Krzysztof Czarnecki for inspiring discussions as well as Gregor Kiczales and the AspectJ team for their support and advices.

References

1. Batory, D., Geraci, B. J.: Composition Validation and Subjectivity in GenVoca Generators. In: IEEE Transactions on Software Engineering. Special issue on Software Reuse (1997) 67 – 82
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. John Wiley, Chichester, New York, Brisbane, Toronto, Singapore (1996)
3. Coulouris, G. F., Dollimore, J.: Distributed Systems: Concepts and Design. Addison-Wesley (1988)
4. Czarnecki, K., Eisenecker, U. W.: Synthesizing Objects. In: Proceedings of ECOOP'99, 13th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science LNCS 1628. Springer-Verlag, June (1999) 18 – 42
5. Eisenecker, U. W.: Generative Programming GP with C++. In: Mössenböck, H.-P. (ed.): Proceedings of Modular Programming Languages JMLC97. Lecture Notes in Computer Science LNCS 1204. Springer-Verlag, Heidelberg, March (1997) 351–365
6. Fayad, M. E., Schmidt, D. C.: Object-Oriented Application Frameworks. In: Communications of the ACM. vol. 40, no. 10, October (1997) 32 – 38
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstractions and Reuse of Object-Oriented Software. Addison-Wesley, Reading, MA (1994)
8. Gough, J., Klaeren, H.: Executable Assertions and Separate Compilation. In: Mössenböck, H.-P. (ed.): Proceedings Joint Modular Languages Conference. Lecture Notes in Computer Science LNCS 1204. Springer-Verlag, Berlin Heidelberg New York (1997) 41–52
9. Guerraoui, R., Fayad, M.: OO Distributed Programming Is Not Distributed OO Programming. In: Communications of the ACM. vol. 42, no. 4, April (1999) 101 – 104

10. Hauck, F., Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M.: AspectIX: A Middleware for Aspect-Oriented Programming. In: Object-Oriented Technology, ECOOP98 Workshop Reader. LNCS 1543. Finland. Springer-Verlag (1998) 426 – 427
11. Kenens, P., Michiels, S., Matthijs, F., Robben, B., Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P.: An AOP Case with Static and Dynamic Aspects. In: Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98. Brussels (1998)
12. Kiczales, G.: Private Communication. June, 1999
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming In: Proceedings of ECOOP'97, European Conference on Object-Oriented Programming. Lecture Notes in Computer Science LNCS 1241. Springer-Verlag, June (1997)
14. Lopes, C. V., Kiczales, G.: Recent Developments in AspectJ. In: Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98. Brussels (1998)
15. Lunau, C. P.: Is Composition of Metaobjects = Aspect Oriented Programming. In: Proceedings of the Aspect-Oriented Programming Workshop at ECOOP98. Brussels (1998)
16. Mowbray, T. J., Malveau, R.C.: Corba Design Patterns. John Wiley and Sons, Inc., New York, Chichester, Weinheim, Toronto, Singapore, Brisbane (1997)
17. Object Management Group: CORBAservices: Common Object Services Specification. December (1998)
18. Object Management Group: The Common Object Request Broker: Architecture and Specification. February (1998)
19. Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, Reading MA (1994)
20. Pree,W.: Komponenten basierte Softwareentwicklung mit Frameworks. dpunkt, Heidelberg (1997)
21. Purao, S., Jain, H., Nazareth, D.: Effective Distribution of Object-Oriented Applications. In: Communications of the ACM. vol. 41, no. 8, August (1998) 100 – 108
22. Schmidt, D.C.: Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. In: Communication of the ACM. Special Issue on Object-Oriented Experiences. vol. 38, no. 10, Brisbane, Australia (1995)
23. Schmidt, D. C.: ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. `http://www.cs.wustl.edu/~schmidt/ACE.html`. New Orleans (1999)
24. Siegel,J.: OMG Overview: CORBA and the OMA in Enterprise Computing. In: Communications of the ACM. vol. 41, no. 10, October (1998) 37 – 43
25. Silva, A. R., Sousa, P., Marques, J. A.: Development of Distributed Applications with Separation of Concerns: In: IEEE Proceedings of the 1995 Asia-Pacific Software Engineering Conference APSEC95. Brisbane, Australia (1995)
26. Sun Microsystems: JavaIDL Documentation. `http://java.sun.com/products/1.2/docs/guide/idl/`, (1999)
27. University of Twente: Homepage of the TRESE Project. The Netherlands. `http://wwwtrese.cs.utwente.nl/`, `http://wwwtrese.cs.utwente.nl/sina/cfom/index.html`, (1999)
28. XEROX Palo Alto Research Center: Homepage of AspectJ. `http://www.parc.xerox.com/aop/aspectj`, (1999)