Managing Complexity In Middleware

Adrian Colyer
IBM UK Limited
Hursley Park, Winchester
England. SO21 2JN
+44 (0)1962 816329
adrian_colyer@uk.ibm.com

Gordon Blair
Computing Department
Lancaster University, Bailrigg
Lancaster, England.LA1 4YR
+44 (0)1524 593809
gordon@comp.lancs.ac.uk

Awais Rashid
Computing Department
Lancaster University, Bailrigg
Lancaster, England. LA1 4YR
+44 (0)1524 592344
marash@comp.lancs.ac.uk

ABSTRACT
Middleware is becoming increasingly complex, and this complexity is at odds with one of middleware's key goals – to make it easier to build distributed systems. A new emphasis on simplicity, componentization and application-middleware independence is required to redress the situation. Aspect-oriented software development techniques hold great promise in helping to meet these challenges, though the large scale of many middleware development projects raises additional requirements that must be met.

1. INTRODUCTION
"There is literally no sensible, economic way to develop distributed applications without middleware services."
– Richard Schreiber, 1995 [1]

Enterprise applications depend on distributed systems, and therefore on middleware. Within the enterprise, distributed systems are used to provide high levels of availability and scalability, to physically separate components for security reasons, to cope with the geographic spread of multi-national corporations, and to exploit the price-performance characteristics of PC and Unix based workstation clusters. Distributed systems also arise naturally through mergers and acquisitions, and business-to-business applications that span organizational boundaries.

Building distributed systems directly on top of networked operating systems is expensive, error-prone and difficult [2], therefore corporate developers rely on middleware, whose primary purpose is to make it easier to build, deploy and operate distributed applications. Middleware makes building distributed systems easier by resolving heterogeneity, providing transparency of various kinds, and by providing qualities of service.

In section 2, we argue that middleware itself is becoming increasingly complex as we strive to build ever more sophisticated distributed systems. If left unchecked, this trend will leave us facing the same set of problems that middleware was intended to solve in the first place – building distributed systems for enterprise applications will be too complex.

Section 3 sets a direction for the development of future middleware platforms based on simplicity, independence of applications from middleware, and componentization. In section 4 we discuss the application of aspect-oriented software development (AOSD) techniques to meet these goals, and in section 5 we discuss the implications of the large scale of many middleware development projects on AOSD. Section 6 concludes and provides a brief summary of related work.

2. MIDDLEWARE COMPLEXITY
"There is already too much diversity of middleware for many customers and application developers to cope with ... the complexity of current middleware is untenable over the long term" – Philip Bernstein, 1996 [3].

Middleware resolves heterogeneity and provides transparency in order to make it simpler to build distributed systems. Yet middleware itself is becoming increasingly complex – there are many heterogeneous middleware environments that need to be integrated, and the use of middleware is not transparent to the application developer.

This complexity is driven by rich feature sets and feature interactions both within and across middleware products, the need to support ever more diverse environments, and the introduction of (needed) more sophisticated capabilities that threaten to reduce transparency further.

2.1 Feature Complexity
Many modern middleware products are rich in features, each feature independently justifiable for sound business reasons. Current state-of-the-practice is to roll these features into a large, monolithic product. This can result in significant complexity and confusion for the application developer working on the middleware platform. Often there are several ways of achieving the same goal with no obvious rationale for choosing between them, and the sheer number of features to learn and investigate can be overwhelming. Consider as an example the Java™2 Platform Enterprise Edition (J2EE™) [4]: J2EE provides a wealth of APIs and services, the cornerstone of which are Enterprise JavaBeans™(EJB™), JavaServer Pages™(JSP™), and Servlets. Mastering the art of building EJB based systems alone can be quite a challenge [5].

2.2 Portfolio Complexity
As well as complexity within a single middleware product, there is additional significant complexity when multiple middleware products are used. The inevitable overlap in capabilities caused by their large feature sets creates additional duplicate means of achieving an end, and hence further developer confusion.

Nearly all large enterprises contain a broad mix of middleware products and home-grown capabilities acquired through the processes of time, business and mission growth, and even mergers and acquisitions. The resulting computing facilities are hard to integrate, and even harder to operate and administer.

2.3 New Requirements

The need to support increasingly heterogeneous environments also drives middleware complexity. Pervasive (or ubiquitous) computing brings challenges due to widely varying device formats and capabilities, unreliable network connections, disconnected operation considerations, and mobility of devices (nomadic devices and ad-hoc networking) [6]. Pushing into new markets, such as taking an enterprise product into the small-medium business (SMB) segment, places new emphasis on existing requirements, and creates entirely new requirements too. The integration of business applications across business boundaries (B2B applications) unites very diverse operating environments across independent domains of control. New transaction models and interaction protocols are needed to deal with this additional complexity.

Meanwhile, middleware research continues to push the boundaries of current capabilities, for example in support of very large scale systems [7, 8] and reflective middleware [9]. A common trend in this research is to give the programmer more influence over the behavior of the middleware. The consequence is that more of the middleware becomes visible to the programmer, and some aspects of distribution and heterogeneity become less transparent [2].

3. MANAGING COMPLEXITY

"For the next several years, corporate buyers will…look for technologies that address business problems directly; provide near-term return on investment, and improve customer acquisition and retention, cost-cutting, revenue or profits." – FORTUNE, March 18th 2002 [10].

The rise in middleware complexity comes at a time when technology discussions are moving from the IT department to the boardroom. Here the debate centers on solutions to business problems, not technology platforms. Business solutions need to be built and deployed as speedily as possible in order to remain competitive; this calls for a responsive middleware platform in which mastery and deployment of only those components absolutely necessary for the task in hand is required.

To take middleware forwards, we need to focus on simplicity instead of complexity, on componentization and configuration instead of monolithic construction, and on loosening the ties between an application and the middleware platform it executes on. All three of these goals are considered from the perspective of the user of the middleware.

3.1 Simplicity

Simplicity is required in the tools used to build applications for a middleware platform, in the programming model exposed by the middleware, in the administration and configuration of the middleware, and in the operation of the middleware.

For middleware programming models and tools, the goal is to make the use of middleware as transparent as possible, so that enterprise application developers spend the majority of their time working in the business application domain focusing on the business problem at hand. Several authors have shown that achieving full transparency of middleware is not possible [2, 11] as some aspects of distribution such as network latency and end-to-end correctness cannot be fully hidden. Approaches that have been, or are being, tried to get as close as possible to this goal include 4GLs, modeling, code generation and declarative specification.

Simplicity in administration and operation requires systems that are self-configuring, self-optimizing, self-protecting and self-healing. IBM® calls such systems "autonomic" [12]. Internally, such a system may well be more sophisticated and more complex than current generation middleware, but the system externals should be much simpler.

3.2 Componentization

Users need to be able to subset the full capabilities of a middleware platform in order to select a feature and footprint combination suitable to the task in hand. The large size of middleware products points to a desperate need for greater componentization of middleware to support this goal. Instead of monolithic middleware products, we need a sophisticated middleware *production line*[1] that can assemble components on demand to provide a given set of capabilities within a given operating environment. Advanced platforms may also permit runtime component selection and configuration.

Software engineering tools to analyze, separate, manage and compose the rich set of features and feature interactions typically found in middleware are immature or non-existent. The problem is hampered by middleware's performance sensitivity, which makes developers wary of large frameworks with layers of indirection.

3.3 Application-Middleware Independence

Middleware platforms continue to change and evolve, and large enterprises tend to acquire plenty of them [13]. Loosening the dependence of a given application on a particular middleware platform or version of a middleware product is good for both application developers and middleware vendors. Application developers can preserve their investment across multiple middleware platform iterations, and middleware vendors can lower the version-to-version migration or competitive win-back costs.

Application-middleware independence necessitates that much of the detail and complexity of middleware is hidden from the application.

4. THE ROLE OF AOSD

Section 3 described *what* needs to be done, but said nothing about *how* the requirements could be met. In this section we describe how a combination of aspect and component based techniques may be employed to that end. We assume that the reader is already familiar with aspect-oriented concepts.

4.1 Simplicity

Declarative specification is one of the most promising approaches for achieving simplicity in programming models. In this section we argue that aspect-oriented software

[1] The sophistication in the *production line* is its ability to create variants of the middleware. The resulting set of products form a *product line*, which may or may not be sophisticated. In the extreme case, the production line may produce product variants tuned for individual customers.

development is a natural fit with a declarative style, aiding simplicity by furthering its application

Declarative specification separates the declaration of the (middleware) services required from the implementation of the business logic that requires them. Declarative specifications are typically honoured by the middleware through some or all of application development-time code generation, deployment-time code generation, and runtime configuration and interpretation. The following Xdoclet [14] fragment is an example of declarative specification through attribute-oriented programming. It declares that a transaction is required to execute the method being commented on:

```
/**
 * … other comments omitted for brevity
 * @ejb.transaction
 *   type="Required"
 */
```

Attributed programming in .NET® [15], and *explicit programming* as exemplified by the ELIDE system [16] work in a similar fashion. In a post [17] to the AspectJ [18] users mailing list, Gregor Kiczales describes the techniques as a form of "early-AOP:"

"…(the) approach requires tagging methods and classes where aspects might apply with attributes. I believe the approach [they outline] can perhaps be called early AOP, but it is missing one of the most critical properties of all other AOP systems, and this significantly limits its power. I call it early AOP because when some people first hear about AOP, this is one of the first mechanisms they propose to achieve it."

In a full aspect-oriented approach, instead of explicitly tagging each element that is to acquire a certain property, we can separate the concerns and encapsulate (for example) the transaction policy of the system into a single unit. The elements that are to acquire transaction semantics are not individually tagged. Thus we can view, maintain, and add or remove the transaction policy of our system as a single unit. Clearly this treatment can be applied to any attribute already separated from the user application through declarative specification (it is not the intent of this paper to discuss the wisdom or otherwise of declaratively specifying transactions [19]).

Declarative specification, when coupled with an ability to interpret declarations and apply appropriate aspects to an element at either class-load time or run-time, can remove the need for code generation completely. JBOSS [20] uses this approach to apply advice to EJBs in the form of interceptors [21]. A more fully fledged form of aspect-oriented programming is promised for the JBOSS 4.0 release, which will permit interceptors to be added to methods, constructors and fields of not just EJBs, but any Java object.

In situations requiring more sophisticated capabilities than can be provided by interceptors alone (such as introducing new methods, fields or parent classes to an application domain object), aspect-orientation allows the generation of code that fully separates the concerns of the middleware from the pure application concerns. The sample code for a stateless session bean from Sun's online EJB tutorial [22] is 90 lines long, and contains only 6 lines of business application logic. It is typical of the kind of template implementation that may be generated by an EJB tool. Using aspect-oriented software development techniques such as those offered by AspectJ or Hyper/J [23], the application class can simply become:

```
public class DemoBean {
  public String demoSelect( ) throws
RemoteException {
    return( "hello world" );
  }
}
```

The application class is not cluttered with EJB-specifics. An aspect-aware EJB tool could then generate an accompanying aspect (shown here as an AspectJ example) that might look something like this:

```
aspect DemoBeanEJB {
  declare parents:
     DemoBean implements SessionBean;

  static final boolean
     DemoBean.verbose = true;

  private transient SessionContext
     DemoBean.ctx;
  …
  public void DemoBean.ejbActivate( ) {
    if (verbose) {
      System.out.println(
         "ejbActivate called" );
    }
  }
  // etc.
}
```

This clear separation between middleware specific concerns and the business logic simplifies the task of application development and greatly improves application-middleware independence. It is also a tremendous advantage for modeling tools supporting round-tripping since the user-written code and generated middleware code are cleanly separated – allowing for safe regeneration of the middleware code without any concern for loss of user updates.

The Java Aspect Components project (JAC) [24] seeks to take these ideas to their logical conclusion, replacing EJBs altogether with simple Java objects and aspect components that can be dynamically plugged into the system at runtime.

4.2 Componentization

Many research and commercial projects are investigating the componentization of middleware – JBOSS for example has a "super-server" architecture with componentization and configuration handled through a JMX (Java Management Extension) spine. TAO [25] is a CORBA implementation focused on high-performance and real-time scenarios. It is built on ACE [26], which can automate system configuration and re-configuration by dynamically linking services into applications at runtime. The Eclipse IDE [27] demonstrates excellent componentization through its model of features,

plugins and extension points[2]. In this section we focus explicitly on the application of aspect-oriented techniques to facilitate and further these efforts.

Aspect-oriented software development techniques provide us with new ways to modularize and encapsulate concerns that were previously entangled (or scattered) across multiple other concerns. Until a concern is encapsulated, it is very difficult to add, remove or replace that concern in a middleware system. Therefore by improving our ability to modularize, aspect-oriented techniques improve our ability to factor a middleware system into components. In a study conducted at IBM's Hursley Laboratory for example, we have shown that tracing, logging, first-failure data capture and performance monitoring instrumentation in a commercial middleware system were all amenable to modularization via aspect-oriented programming techniques [28]. Previously these concerns were scattered throughout the system. Similarly, [29] shows the use of aspect-oriented techniques within a database system for concern encapsulation.

Frank Hunleth has studied the use of AspectJ for feature and footprint management in middleware systems, using aspects to introduce features incrementally and as independently as possible [30, 31]. Lasagne [32] uses aspect-oriented software development to construct customizable middleware and distributed services, focusing on context-sensitive customizations.

A promising direction seems to be the use of several aspect-oriented techniques in combination to factor the middleware in multiple dimensions: a Composition Filter [33] like approach for simple interception based strategies, an aspect-oriented language such as AspectJ for cross-cutting concerns, sophisticated composition models such as those offered by Hyper/J for large-scale feature and system integration, and adaptive programming techniques such as those offered by DemeterJ [33] for structure-shy object relationship traversals. Such a hybrid approach was first proposed in [34].

4.3 Application-Middleware Independence

Current (non-AO) approaches to application-middleware independence such as the OMG's Model Driven Architecture (MDA) [13] rely mainly on abstraction and thus either reveal complexity and tight coupling at the more concrete levels (generated code or platform-specific models), or are limited in their application by an inability to express often needed details (declarative specifications and 4GLs). By separating middleware concerns from application domain concerns at all levels of abstraction, using techniques such as those promoted by aspect-oriented software development, we retain the ability to fully express an application's middleware requirements at the needed level of detail without adversely affecting coupling. Section 4.1 illustrated the use of aspect-oriented software development techniques to separate application and middleware concerns at the implementation level. The combination of abstraction and separation in the binding of applications to middleware is illustrated in Figure 1.

Quadrant A shows an abstract model of the business application with no middleware details. Quadrant B adds an abstract model of middleware requirements. Quadrant C shows a typical concrete middleware based application with business and middleware concerns entwined. Quadrant D shows a concrete middleware based application with business and middleware concerns separated.

Many applications begin and end life in quadrant C. Code generation and simple application domain modeling follows the path A → C. MDA attempts to introduce the path A → B → C, although the separation in quadrant B is not as clean as depicted. Combining abstraction and separation gives us the new endpoint D, and development path A → B → D. It can be clearly seen how the concrete, separated system in quadrant D can handle evolution or replacement of the middleware portion much more gracefully than the system in quadrant C.

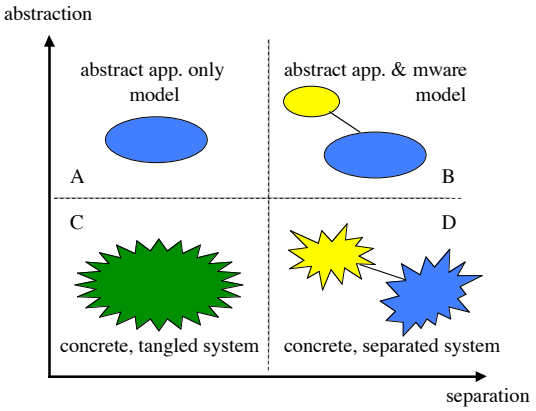

Figure 1: Abstraction and Separation in Middleware

The adaptive programming approach supported by Demeter provides another useful tool in the quest for application-middleware independence. It supports "structure-shy" traversal strategies that separate behaviour from structure, and hence allows inter-position of middleware components (such as wrappers and facades) in a manner transparent to the application.

5. THE CHALLENGE OF SCALE

As discussed in section 2.1, middleware products tend to be large – for example, IBM's WebSphere® Application Server comprises many thousands of Java classes and is developed and maintained by hundreds of staff. Many concerns in the middleware system are looked after by entire teams, and apply broadly to the independently developed work of multiple other teams. It is therefore essential to be able to specify the broad policy pertaining to a concern and where it should be applied, and then independently permit special cases (exceptions or additions to the general policy) to be specified by the owners of the affected concerns.

We envisage the production line of section 3.2 working by configuring *and composing* components to produce the required variants. A single level of configuration or composition is not tenable for software of this complexity (both because the configuration file itself would be overwhelming, and because updating the file under version control would bottleneck parallel development streams).

[2] Application development tools are an important part of a middleware platform.

Instead a "fractal" approach is required, whereby any given component may be composed of multiple sub-components, which in turn are composed of multiple sub-components and so on. The decomposition terminates with primitive (atomic) software units as determined by the metamodel of the programming language and runtime in use. Component composition is hidden from users of that component. Note that this philosophy requires that we distinguish carefully between aspect-oriented techniques that create or compose new units of existing (meta)types, and those that introduce new meta-types.

Fitting neatly with the fractal view of software composition is the observation that middleware products are not simply "compiled," but rather "built" on a production line involving many stages from initial compilation through to deployment and automated system testing. One component depends on other components, and the build infrastructure ensures that dependencies are built (compiled, assembled, composed) before the dependent component. This points to the need for the strong integration of AO-techniques for middleware with build environments, the primary of which is Apache Ant[35].

6. RELATED WORK

In addition to those tools and techniques already mentioned, there are many other active research projects, of which a few are highlighted here. In general, the emphasis of these projects is on enhancing the capabilities of an infrastructure platform, rather than the (more internally oriented) use of aspect-orientation to simplify the construction and presentation of existing capabilities.

DAOP [36] is a dynamic aspect-oriented platform providing a composition mechanism for integrating aspects and components dynamically at runtime. The DADO [37] (distributed aspects for distributed objects) project helps program crosscutting features in heterogeneous environments. Choi [38] shows how aspect-orientation can be used to build an open extensible container with EJB facilities. Duclos [39] extends the concepts in EJB and the CORBA Component Model to fully separate container services from business logic. In contrast, Kim [40] discusses the relevance of AOP within an existing EJB container.

7. SUMMARY

Enterprise computing requires distributed systems, even though distributed systems introduce considerable complexity into application development and system management. Middleware facilitates the building of distributed systems, resolving many of the lower-level problems associated with distribution and heterogeneity.

Now middleware itself is suffering a crisis of complexity and heterogeneity. This paper presents an analysis of the causes of middleware complexity, and sets a direction to return to the original focus of middleware – making distributed systems easier to build. To achieve this end the middleware community needs to focus on:

- Simplicity of application development, administration, and operation.
- Separating middleware into pluggable components that can be put together in middleware production lines to more precisely meet the needs of a given application running in a given environment.
- Loosening the ties between an application and the middleware platform(s), products, and product versions that it executes on.

We have shown that aspect-oriented software development is well suited to helping middleware address these challenges. AOSD is a natural fit with a declarative specification style, and aids in the drive for simplicity by furthering its application. AOSD also provides new mechanisms to compose software artifacts, allowing us to separate and encapsulate concerns that previously could not be easily separated. It can therefore facilitate the separation of middleware components and their subsequent re-composition to meet the needs of a given application or environment. Finally, AOSD can also help separate middleware details from application domain concerns, improving application-middleware independence.

Future directions for this work include the evolution of aspect-oriented techniques to meet the challenges described in section 5, when applied to componentization of large-scale commercial middleware. Work is also underway to investigate the role of AOSD within the OMG's MDA, which shares a common goal in application-middleware independence. This includes using aspect-oriented techniques to facilitate generation, from declarative specifications in models, of cleanly separated code implementing middleware concerns.

8. ACKNOWLEDGMENTS

IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries or both.

Microsoft and .Net are registered trademarks of Microsoft Corporation in the United States, other countries or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

9. REFERENCES

1. Schreiber, R., *Middleware Demystified*, in *Datamation*. 1995. p. 41-45.
2. Emmerich, W., *Software Engineering and Middleware; A Roadmap*, in *The Future of Software Engineering 2000*, A. Finklestein, Editor. 2000, 22nd International Conference on Software Engineering. p. 117-129.
3. Bernstein, P., *Middleware: A model for distributed systems services*, in *Communications of the ACM*. 1996. p. 86-98.
4. *Java 2 Enterprise Edition (J2EE)*, Sun Microsystems: http://www.java.sun.com/j2ee.
5. Zeichek, A., *WebSphere Goes Lite (sidebar)*, in *Software Development Times*. 2002: http://www.sdtimes.com/news/068/story1.htm.
6. Geihs, K., *Middleware Challenges Ahead.* IEEE Computer, 2001. **34**(6): p. 24-31.
7. van-Steen, M., P. Homburg, and A. Tanenbaum, *Globe: A Wide-Area Distributed System.* IEEE Concurrency, 1999. **7**(1): p. 104-109.

8. Vaughan-Nichols, S.J., *Developing the Distributed Computing OS.* IEEE Computer, 2002. **35**(9): p. 19-21.
9. Coulson, G., *What is Reflective Middleware?*, in *IEEE Distributed Systems Online*. 2002: http://dsonline.computer.org/middleware/RMarticle1.htm.
10. Kirkpatrick, D., *Beyond buzzwords*, in *FORTUNE*. March 18, 2002.
11. Saltzer, J.H., D.P. Reed, and D.D. Clark, *End-to-End Arguments in System Design.* ACM Transactions on Computer Systems, 1984. **2**(4): p. 277-88.
12. *Autonomic Computing*, IBM: http://www.research.ibm.com/autonomic.
13. *Model Driven Architecture*, OMG: http://www.omg.org/mda/.
14. *XDoclet: Attribute Oriented Programming*, The XDoclet team: http://xdoclet.sourceforge.net.
15. Shukla, D., S. Fell, and C. Sells, *Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse*, in *MSDN Magazine, March*. 2002.
16. Bryant, A., et al. *Explicit Programming*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM press.
17. Kiczales, G., *AOP .net?* 2002: post to users@aspectj.org, http://aspectj.org/pipermail/users/2002/001846.html.
18. Kiczales, G., et al. *Aspect-oriented programming*. in *ECOOP '97 - Object Oriented Programming 11th Europeann Conference*. 1997. Jyvaskyla, Finland: Springer-Verlag.
19. Kienzle, J. and R. Guerraoui. *AOP: Does it Make Sense? The Case of Concurrency and Failures.* in *ECOOP 2002 - Object-Oriented Programming*. 2002. Malaga, Spain: Springer.
20. *JBOSS Home Page*, JBOSS Group: http://www.jboss.org.
21. Fleury, M., *BLUE: "Why I Love EJBs"*. 2002, JBOSS: http://www.jboss.org/blue.pdf.
22. *Online EJB Tutorial: Writing the Enterprise JavaBean class*, Sun Microsystems: http://developer.java.sun.com/developer/onlineTraining/Beans/EJBTutorial/step4.html.
23. Ossher, H. and P. Tarr, *Using Multidimensional Separation of Concerns to (re)shape Evolving Software.* Communications of the ACM, 2001. **44**(10): p. 43-49.
24. Pawlak, R., et al., *JAC: A flexible solution for aspect-oriented programming in Java.* Reflection 2001, 2001. **LNCS 2192**: p. 1-24.
25. Schmidt, D., *Applying Patterns to Develop Extensible ORB Middleware.* IEEE Communications Magazine, 1999(April).
26. Schmidt, D. *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*. in *12th Annual Sun Users Group Conference*. 1994. San Francisco, CA.
27. Amsden, J. and A. Irvine, *Your First Plug-In*. 2002: http://www.eclipse.org/articles.
28. Bodkin, R., A. Colyer, and J. Hugunin. *Applying AOP for Middleware Platform Independence*. in *Practitioner Reports, 2nd International Conference on AOSD - To Appear*. 2003. Boston, MA.
29. Rashid, A. and P. Sawyer, *Aspect-orientation and database systems: an effective customisation approach.* IEE Proceedings - Software, 2001. **148**(5): p. 156-164.
30. Hunleth, F. and R. Cytron. *Footprint and Feature Management Using Aspect Oriented Programming Techniques*. in *LCTES 02*. 2002. Berlin, Germany: ACM.
31. Hunleth, F., R. Cytron, and C. Gill. *Building Customizable Middleware using Aspect Oriented Programming*. in *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. 2001. Tampa, Florida.
32. Truyen, E., et al. *Dynamic and Selective Combination of Extensions in Component-based Applications*. in *Proceedings of the 23rd International Conference on Software Engineering*. 2001. Toronto, Canada.
33. Bergmans, L. and M. Aksit, *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, 2001. **44**(10): p. 51-57.
34. Rashid, A. *A Hybrid Approach to Separation of Concerns: The Story of SADES*. in *Reflection 2001*. 2001. Kyoto, Japan: LNCS.
35. *Apache Ant*, The Apache Jakarta Project: http://jakarta.apache.org/ant.
36. Pinto, M., L. Fuentes, and J.M. Troya, *DAOP-ADL: An Architecture Description Language for Dynamic Aspect-Oriented Development.*
37. Wohlstadter, E., S. Jackson, and P. Devanbu, *DADO: Enhancing Middleware to support cross-cutting features in distributed, heterogeneous systems.* To Appear.
38. Choi, J.P. *Aspect oriented programming with Enterprise JavaBeans*. in *Fourth International Enterprise Distributed Objects Computing Conference*. 2000. Makuhari, Japan: IEEE Computer Soc.
39. Duclos, F., J. Estublier, and P. Morat. *Describing and Using Non Functional Aspects in Component Based Applications*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM Press.
40. Kim, H. and S. Clarke, *The relevance of AOP to an Applications Programmer in an EJB environment*, in *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*. 2002.