



Survey of Aspect-oriented Middleware

ABSTRACT

Distributed systems software is increasing in its size and complexity, making its development highly problematic. To aid in the construction of large scale distributed systems, many software developers have adopted middleware approaches. However, middleware itself is becoming increasingly complex; so complex in fact that it threatens to undermine one of its key aims – to simplify the construction of distributed systems. In this report we survey the current state of the art middleware approaches and AOSD middleware approaches. Additionally, we report on how AOSD has been utilised in the structuring of middleware. Our conclusions lead us to believe that middleware and AOSD complement each other naturally, in that each plays to one another's strengths while obviating many of the problems inherent in traditional middleware approaches.

Document ID:	AOSD-Europe-ULANC-10
Deliverable	
/Milestone No:	D8
Work-package No:	WP9
Type:	Survey
Status:	FINAL
Version:	1.0
Date:	14 June 2005
Author(s):	Neil Loughran and Nikos Parlavantzas (ULANCS), Monica Pinto, Lidia Fuentes Fernández and Pablo Sánchez (UMA), Matthew Webster and Adrian Colyer (IBM)

Table of Contents

1. Introduction.....	7
1.1 Structure of the Report.....	8
2. Overview of Technologies for Supporting the Management and Evolution of Crosscutting and Systemic Concerns Relating to Middleware.....	9
2.1 Requirements for Middleware	9
2.1.1 Customisability	10
2.1.1.1 Customisability of binding management	10
2.1.1.2 Customisation of resource management	11
2.1.1.3 Flexibility	12
2.1.1.4 Ease of customisation	13
2.1.1.5 Reliability.....	13
2.1.2 Usefulness	14
2.1.2.1 Ease of use	14
2.1.2.2 Performance	14
2.2 Commercial Middleware Platforms.....	15
2.2.1 CORBA.....	15
2.2.2 J2EE and Jini.....	17
2.2.3 COM and .Net.....	19
2.3 Historical Overview of Predecessors of Aspect-oriented Middleware Technologies.....	20
2.3.1 Separation of Concerns	20
2.3.2 Design Patterns	20
2.3.3 Reflection.....	21
2.3.4 Component Based Software Engineering	21
2.3.5 Object Oriented Frameworks	21
2.3.6 Compositional Adaptation	22
2.4 Research Middleware Platforms	22
2.4.1 FlexiNet.....	22
2.4.2 Jonathan	23
2.4.3 OpenORB.....	24
2.4.4 2K, dynamicTAO, and UIC	26
2.4.5 Customisation of ORBs by Jørgensen et al.....	27
2.4.6 Multe-Orb	29
2.4.7 K-Components	30
2.4.8 Quartz.....	31
2.4.9 QoS-enabled middleware.....	32
2.5 Evaluation Overview	32
2.5.1 Quantitative view	32
2.5.2 Discussion	35
2.6 Summary	36
3. Overview of Middleware Technologies providing an Aspect Oriented Programming Model	37
3.1 Introduction.....	37
3.2 Comparison Criteria.....	38
3.3 AO4BPEL	39
3.3.1 Programming Model	39
3.3.2 Platform Infrastructure and Services	40
3.3.3 Current Status of Development.....	42
3.3.4 Evaluation	43

3.4	AspectJ2EE.....	44
3.4.1	Programming Model	44
3.4.2	Platform Infrastructure and Services	45
3.4.3	Current State of Development	46
3.4.4	Evaluation	46
3.5	AspectWerkz.....	47
3.5.1	Programming Model	47
3.5.2	Platform Infrastructure and Services	49
3.5.3	Current Status of Development.....	50
3.5.4	Evaluation	51
3.6	CAM/DAOP	52
3.6.1	Programming Model	52
3.6.2	Platform Infrastructure and Services	55
3.6.3	Current Status of Development.....	57
3.6.4	Evaluation	58
3.7	JAC	59
3.7.1	Programming Model	59
3.7.2	Platform Infrastructure and Services	62
3.7.3	Current Status of Development.....	64
3.7.4	Evaluation	64
3.8	JBoss AOP	65
3.8.1	Programming Model	65
3.8.2	Platform Infrastructure and Services	67
3.8.3	Current State of Development	68
3.8.4	Evaluation	69
3.9	Lasagne	70
3.9.1	Programming model.....	70
3.9.2	Platform Infrastructure and Services	72
3.9.3	Current State of Development	74
3.9.4	Evaluation	75
3.10	PRISMA.....	76
3.10.1	Programming Model	76
3.10.2	Platform Infrastructure and Services	78
3.10.3	Current Status of Development.....	79
3.10.4	Evaluation	79
3.11	PROSE/MIDAS	80
3.11.1	Programming Model	80
3.11.2	Platform Infrastructure and Services	82
3.11.3	Current Status of Development.....	84
3.11.4	Evaluation	84
3.12	Spring AOP.....	85
3.12.1	Programming Model	86
3.12.2	Platform Infrastructure and Services	87
3.12.3	Current Status of Development.....	88
3.12.4	Evaluation	89
3.13	WSML/JAsCo.....	89
3.13.1	Programming Model	90
3.13.2	Platform Infrastructure and Services	92
3.13.3	Current Status of Development.....	93
3.13.4	Evaluation	94

3.14	Evaluation Overview	94
3.14.1	Evaluation According to General Criteria	95
3.14.1.1	Quantitative view	95
3.14.1.2	Discussion	95
3.14.2	Evaluation According to AO Specific Criteria	97
3.14.2.1	Quantitative view	97
3.14.2.2	Discussion	99
3.15	Summary	101
4.	Using AOSD to Structure Middleware	102
4.1	Host-Infrastructure middleware	102
4.2	Distribution Middleware	104
4.3	Middleware Services	105
4.4	Domain-Specific Middleware Services	105
4.5	Summary	106
5.	Conclusions	107
6.	Acknowledgements	109
	References	110

List of Figures

Figure 1. Hierarchical structure for middleware evaluation criteria.....	10
Figure 2. Architecture of the AO4BPEL Web Service Composition System	41
Figure 3. AO4BPEL middleware services.....	42
Figure 4. Integration of AspectWerkz in an Application Server	50
Figure 5. Architecture and services of the DAOP platform.....	55
Figure 6. JAC wrapper instantiation mode	61
Figure 7. The JAC architecture	63
Figure 8. JAC distributed joinpoints and aspects.....	64
Figure 9. Architecture of the JBoss Application Server	68
Figure 10. High level functional decomposition of the runtime system.....	72
Figure 11. Runtime weaving mechanism in Lasagne	74
Figure 12. PRISMA architecture	78
Figure 13. MIDAS service community.....	81
Figure 14. MIDAS architecture	82
Figure 15. PROSE architecture.....	84
Figure 16. Architecture of the Web Services Management Layer.....	93

List of Tables

TABLE 1. Middleware platform comparison	34
TABLE 2. Middleware for AOSD platform comparison with respect to general criteria	95
TABLE 3. Middleware for AOSD platform comparison with respect to AOSD criteria	98

1. Introduction

Distributed systems software is increasing in its size and complexity. While computing resources, processing power and network bandwidth have all increased by orders of magnitude over the last decade; the construction of large scale distributed software systems remains highly problematic. Among the driving technologies that contribute to this complexity are heterogeneous networks, sophisticated enterprise applications, and mobile and ubiquitous computing infrastructures. Additionally, the needs of end users are constantly evolving as new requirements emerge. Evolving business domains such as ecommerce and banking are highly volatile and require high availability, thus requiring more dynamic forms of configuration and adaptation. Economic factors such as minimising costs and time to market have also become increasingly important as organisations search for new ways to become more productive and competitive in the marketplace.

To aid in the construction of large scale distributed systems, many software developers have adopted *middleware approaches*. Middleware facilitates the development of distributed software systems by accommodating heterogeneity, hiding distribution details, and providing a set of common and domain specific services.

However, as [1] points out, middleware itself is becoming increasingly complex; so complex in fact that it threatens to undermine one of its key aims – to simplify the construction of distributed systems. Additionally, [2] describes that the sheer volume of middleware standards and technologies as being a contribution to this complexity. Middleware particularly suffers from increased complexity when addressing concerns of a crosscutting nature. System wide concerns such as persistence, transactional communication, security, quality of service, and synchronisation, cannot be easily modularised and, therefore, become entangled in the system, thus decreasing understandability and potential for reuse.

AOSD [3] is an emerging software development paradigm which aims to solve the problems of crosscutting in large systems. We believe that middleware and AOSD complement each other naturally in that each plays to one another's strengths while obviating many of the problems inherent in middleware.

In this report, we evaluate state of the art middleware platforms and AO middleware approaches on three key criteria, namely, *flexibility*, *reliability* and *performance*. While we detail these criteria extensively in section 2 we comment on them briefly here by way of introduction: *flexibility* is the ability of a platform to accommodate large scale customisation, configuration and extension both statically and dynamically; *reliability* is the ability to eliminate inconsistencies that might be inadvertently introduced by such customisation; and *performance* is the ability of a platform to perform its tasks with adequate speed and minimum consumption of resources.

1.1 Structure of the Report

The structure of this report is as follows: Section 2 discusses precursors to AO middleware technologies and evaluates a selection of commercial and research based middleware platforms. This perspective is important because AO is not the first approach that has attempted to address the general issues raised above. Section 3 then evaluates current AO middleware platforms with respect to our chosen criteria, namely flexibility, reliability and performance. Section 4 then describes industrial experience in using AOSD to structure middleware. Finally, section 5 presents our conclusions.

2. Overview of Technologies for Supporting the Management and Evolution of Crosscutting and Systemic Concerns Relating to Middleware

2.1 Requirements for Middleware

The main purpose of middleware platforms is to simplify the construction of distributed systems from a set of requirements. These requirements can generally be classed as either *functional* or *non-functional*. Functional requirements describe what a system should do, or the functions it should perform. Non-functional requirements are concerned with emergent properties of the system such as reliability, response time and other qualities. A key quality requirement on platforms and the main focus of this work is *customisability*, which bears on the range of supported platform customisations, the effort needed for performing them, and the possibility of introducing inconsistencies. Customisability is motivated by the need to accommodate the diversity and variability of a platform's environment. Failure to address adequately this need reduces the value of the platform and shortens its life span.

Apart from customisability, two other quality requirements are selected for guiding platform design and evaluation; namely, *ease of use* and *performance*. Ease of use is selected because it directly contributes to the achievement of the platform's main purpose; that is, simplifying application development. Performance is selected because it usually conflicts with and must be balanced against customisability. Both qualities are essential for the practical acceptance of modifiable platforms by developers.

A middleware platform interacts with external entities playing two roles: *middleware modifiers* and *middleware users*. Middleware modifiers customise platforms; the customisation may involve either providing new functionality (e.g. implementing a protocol plug-in) or configuring existing functionality (e.g. composing a set of protocols to form a protocol stack). Middleware users, on the other hand, call upon the services of a platform to build larger software systems. Note that the distinction between users and modifiers is orthogonal to the distinction between application developers and middleware developers. Indeed, the modifier role can be played by application developers tailoring a platform to the needs of a specific application, and the user role can be played by middleware developers expanding a platform by building upon existing functionality.

Structuring the platform requirements begins with two high-level requirements that correspond to the perspectives of modifiers and users; namely, *customisability* and *usefulness*. These high-level requirements are then refined into more concrete functional and quality requirements that are suitable for driving platform design and evaluation. Figure 1 presents the resulting hierarchical structure of platform requirements (lines indicate refinement, and clouds and rectangles indicate quality and functional requirements respectively), and the following section discusses the requirements and their refinement in detail.

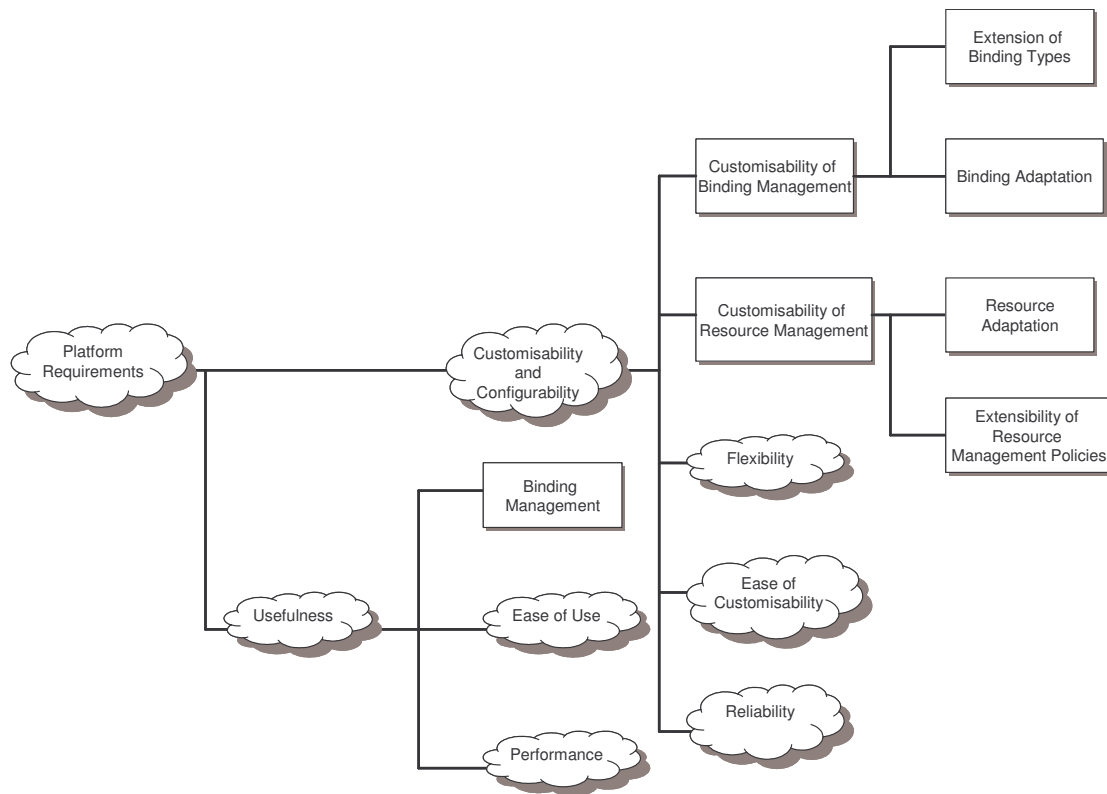


Figure 1. Hierarchical structure for middleware evaluation criteria

2.1.1 Customisability

Customisability refers to a platform’s ability to satisfy the needs of middleware modifiers and is refined into two functional requirements (customisability of communication paradigm and customisation of resource management) and three quality requirements (flexibility, ease of use, and reliability). The former describe essential customisability facilities that must be minimally provided by platforms. The latter constrain and qualify the complete set of provided customisation facilities, not only the minimally required ones. The following sections elaborate on these requirements and their refinements.

2.1.1.1 Customisability of binding management

The primary service offered by middleware platforms to middleware users is interconnecting application elements, that is, establishing bindings between them. The term ‘binding’ is used to abstract over middleware-provided infrastructure that enables application elements to interact in specific ways. As well as supporting functionality for realising bindings, that is, binding management, platforms should support customisation of this functionality. This requirement is refined into two lower-level requirements:

Extension with respect to binding types

Platforms should enable the development and integration of new *binding types*. A binding type (which should not be confused with binding of object types in the

context of object-orientation) describes a specific set of communication patterns together with associated QoS properties. Examples of binding types that have proven to be widely useful include remote method invocation, messaging, continuous media streaming, group communication and shared data spaces. Supporting extension with respect to binding types increases the range of communication needs that can be met by middleware, thus making the platform more valuable to application developers.

To satisfy this requirement, platforms must provide generic facilities in terms of design and run-time support for both using and implementing diverse binding types. Moreover, platforms must support both static and dynamic integration of new binding types. It is worth noting that platforms offering a fixed set of built-in binding types allow implementing new types on top of the built-in ones, but they do not explicitly support this task. Hence, these platforms do not satisfy the extension requirement.

Binding adaptation

Platforms should enable binding adaptation, that is, dynamic inspection and customisation of binding properties. Specifically, customisations should be possible not only at binding establishment time but also during binding-supported interaction. Binding adaptation is required in order to accommodate changing requirements on binding behaviour, which may in turn arise from changes in application needs or infrastructure capabilities. For instance, depending on whether a transmitted video sequence is to be viewed synchronously or recorded for later processing, a video streaming binding may be modified to support different reliability levels. As another example, when the available network bandwidth drops, a remote method invocation binding may be modified to support compression of transferred data.

This requirement does not constrain the specific properties of bindings that are open to adaptation, which are diverse and depend on the associated binding types. For example, customisable properties may take the form of parameters expressing the requested QoS of bindings, such as sample size and sample rate in audio streaming bindings. Or, they may take the form of implementation aspects of bindings, such as transport protocols or marshalling policies in remote method invocation bindings.

2.1.1.2 Customisation of resource management

To develop middleware functionality that exposes high-level abstractions to applications, modifiers invariably need to use underlying resources, such as memory, sockets and kernel level threads. Apart from facilities for resource management, middleware modifiers require facilities for customisation of such functionality. This requirement is refined into two lower level requirements.

Resource adaptation

Platforms should provide facilities for resource adaptation; that is, facilities for both monitoring and controlling the resource usage of activities within a running middleware based system. Resource adaptation is necessary for platforms that target applications with a class of QoS requirements, such as timeliness and capacity, whose significant characteristic is that their satisfaction is influenced by the level of resources provided by the underlying infrastructure. Since resource availability

fluctuates, ensuring that QoS requirements are met requires that the platform incorporates adaptation functionality. It is developing such functionality that is supported by resource adaptation facilities. Specifically, monitoring the resource usage supports making adaptation decisions. For example, knowing the available network bandwidth can drive the selection of an appropriate audio compression algorithm in a platform that supports audio streaming. Controlling the resource usage supports realising adaptation decisions. For example, increasing the CPU time allocated to processing requests can be used to sustain response times in the face of increased network delays. Apart from managing QoS requirements, resource adaptation facilities are useful in many other situations. For example, they can be used for constraining the resource consumption of independently developed and dynamically plugged components. Specifically, the platform could monitor the resources used by a plug-in and decide to remove or replace it if it violates imposed limits. As another example, monitoring can be used for billing middleware users according to resource usage. This could be applied in middleware platforms that simultaneously host multiple applications, such application servers hosting servlets [4].

Extension with respect to resource management policies

Platforms should support the development and integration of new resource management policies. A resource management policy controls how a resource of a specific type is shared. For example, a scheduling policy controls how a processor is multiplexed among jobs (e.g. threads or processes), and a memory allocation policy controls how a memory pool is shared among clients. Policies are designed to satisfy different, sometimes contradictory objectives. As a result, there is an unlimited space of possible policies, which are appropriate under different circumstances. For instance, proportional share scheduling policies, such as lottery scheduling, emphasise flexibility and fairness and are appropriate for real-time applications that must degrade gracefully. On the other hand, reservation-based scheduling policies, such as rate monotonic or earliest deadline first, emphasise guaranteed rate and are best suited for real-time applications that need not degrade gracefully [5]. Similarly, memory allocation policies (e.g. first-fit or best-fit) make different space/time trade-offs and perform best for different memory usage patterns. To deal with this diversity of resource management policies, platforms should therefore support the introduction and flexible configuration of the most suitable policy for each situation.

2.1.1.3 Flexibility

Platforms should be flexible; the more flexible a platform is, the more likely it is that it will be able to absorb a given environmental variation. Intuitively, the flexibility quality requirement assesses the number of all executing variants that are enabled by a platform. To enhance flexibility, platforms should satisfy three constraints. First, they should support customisation both statically (i.e. at design, implementation, and deployment time) and dynamically (i.e. at operating-time). Static flexibility is useful for addressing variations in static properties of the platform environment—that is, properties that remain invariant during operating-time, such as the type of the operating system, the processor speed, or the targeted application domain. Dynamic flexibility, on the other hand, is useful for addressing variations in dynamic properties of the environment, such as the number of members in a group communication

binding or the quality of network connectivity. Importantly, dynamic flexibility enables timely response to such variations without incurring system down-time.

Second, platforms should support extension, that is, the ability to add new functionality. Extension is a particularly desirable type of customisation because, unlike parametric customisation, it ensures that the set of possible platform variants is not a priori fixed, and thus it greatly enhances flexibility.

Finally, platforms should support large-scale customisation; that is, they should support changes affecting large parts of their functionality. Large-scale platform customisation is essential for accommodating the diversity of middleware requirements across application domains and underlying infrastructures. If the platform functionality is viewed as comprising a small number of coarse-grained services, large-scale customisation will involve changing the amount and kind of included services. For instance, a power management service is useful for supporting mobile applications, but not necessary for supporting enterprise computing applications. Therefore, a flexible platform should allow the incorporation of this service to be optional.

2.1.1.4 Ease of customisation

Platforms should make it easy for middleware modifiers to carry out required changes, which may involve providing new functionality or configuring existing functionality. The customisation support provided by platforms can take various forms, including guidelines and design rules, reusable functionality to manage reconfiguration, and common interfaces to enable extension and replacement of middleware components. To enhance ease of customisation, platforms should provide high-level customisation facilities that help bridge the gap between modifiers' goals and the necessary actions for achieving them. For example, a customisation facility that is accessed declaratively, such as selecting between a fixed set of scheduling policies, imposes less workload on modifiers than a facility accessed procedurally, such as implementing a new policy. Moreover, the customisation facilities must be easy to understand and learn. For example, when customisation facilities are structured into consistent interfaces, the effort needed to learn how to use them is reduced. As another example, separating customisation facilities from facilities for using the platform improves the understandability of both.

This requirement conflicts with the flexibility requirement because increasing flexibility inherently increases the number of decisions that have to be faced by modifiers and thus reduces the ease of customisation.

2.1.1.5 Reliability

Platforms should minimise the possibility of inconsistencies introduced by customisations. The extent to which this requirement is satisfied depends on the provided support for reliability. Such support can take various forms. For example, highly effective support can be provided through customisation facilities that validate automatically all changes. Less effective support can involve providing rules and specifications that prevent inconsistencies by design and relying on middleware modifiers to obey them. This requirement conflicts with the flexibility requirement

because increasing the range of allowed changes increases the scope of possible inconsistencies, which typically impairs the platform's ability to manage them.

2.1.2 Usefulness

Usefulness refers to the platform's ability to satisfy the needs of middleware users. For the purpose of this document, usefulness is refined into one functional requirement, that is, supporting binding management, and two quality requirements, that is, ease of use and performance. Binding management is the primary service offered by middleware and therefore it must be minimally provided. Platforms can also offer additional middleware services that simplify application development, such as logging, persistence, and lifecycle management, integrated with each other and binding management in different ways. The ease of use and performance requirements qualify the complete set of middleware services and are examined in the following two sections.

2.1.2.1 Ease of use

Platforms should make it easy for middleware users to employ middleware services in the construction of distributed systems. To increase ease of use, platforms should provide services that present users with a high level of abstraction. For example, middleware services that are accessed declaratively, such as transactions controlled by setting component descriptors, are easier to use than services accessed procedurally, such as transactions controlled by using an explicit transaction API. As another example, consider middleware services for managing bindings, that is, binding types. High-level binding types that hide complex interaction protocols and match closely application needs, such as binding types that support auctions or voting, contribute more to ease of use than low-level binding types, such as remote method invocation. Ease of use is also increased by separating facilities for using middleware from facilities for modifying middleware to the maximum extent feasible.

2.1.2.2 Performance

Platforms should perform their tasks with minimum consumption of resources. The motivation is to ensure that they are usable for building applications with demanding performance needs or applications that operate in resource-constrained environments. To enhance performance, platforms should reduce resource usage that does not directly contribute to meeting middleware users' needs. Platforms induce resource overhead in two main ways. First, overhead is induced by incidental platform dependencies on inefficient languages, APIs, or component models. For example, requiring middleware components to run on a virtual machine, such as the Java virtual machine or .Net CLR, rather than directly on an operating system and hardware introduces a performance penalty due to the indirection. Second, overhead is induced by the platform's responsibility to support customisation. For example, enabling dynamic middleware extension incurs runtime overhead in terms of finding an appropriate component originating in a repository, loading the component into memory, and initialising the component. Due to the overhead for supporting customisation, the performance requirement apparently conflicts with the customisability requirement. On the other hand, the performance of a particular

platform variant can sometimes be improved by applying selected customisations, such as replacing inefficient algorithms, selecting resource management policies that better match actual usage patterns, or removing extraneous functionality. Thus, enhanced customisability can in some circumstances have a positive impact on platform performance. However, since the customisability requirement is covered separately elsewhere, the performance requirement considers only the platform-induced overhead, which remains essentially the same for all platform variants.

2.2 Commercial Middleware Platforms

Here we outline and evaluate middleware technologies widely used in industry, namely, CORBA, J2EE, Jini, COM, and .Net.

2.2.1 CORBA

CORBA is a family of open, vendor-independent standards for building distributed systems published by the Object Management Group (OMG) [6]. At the heart of *CORBA* is the object request broker (ORB) that enables objects to interoperate regardless of their location, the operating system and hardware on which they execute, and the programming language in which they are implemented. Interoperation across programming languages relies on specifying object interfaces in a declarative, language-neutral way using the OMG interface definition language (OMG IDL). Interoperation across different ORB implementations relies on the Internet Inter-ORB Protocol (IIOP). Apart from the ORB, *CORBA* includes:

- common services for facilitating application development (e.g. naming, notification, security, and transaction service)
- extensions for specialised operational environments (e.g. Real-time *CORBA* for real-time systems [7], and Minimum *CORBA* for systems with limited resources [8])
- the *CORBA* component model (CCM), a component technology that supports components distributed over different machines [9]
- domain-specific integration standards (e.g. standards for composing air traffic control or CAD/CAM systems)

CORBA had initially adopted a “black-box” philosophy according to which internal details of middleware services remain unspecified in order to provide greater freedom to vendors of *CORBA*-compliant products. The drawback of this philosophy is that developers are denied the opportunity to modify the middleware functionality, unless they use vendor-specific, non-portable mechanisms. This problem has resulted in a trend towards incorporating increasingly more customisation facilities in *CORBA* standards; the most notable of those facilities are examined next.

The ORB and *CORBA* services support declarative configuration through *policies*, which represent choices affecting their operation. For example, the portable object adapter (POA), the *CORBA* element that intercedes between object implementations and the ORB, defines a policy that controls the threading model; i.e. how threads are used for dispatching object requests. This policy allows selection among three threading models; that is, single-thread, ORB-controlled, and main-thread model. The

portable interceptor's mechanism supports extending the ORB functionality by registering interceptor objects to be invoked at predetermined points of the execution flow; namely, when requests and replies are sent and received, and when object references are created. The configuration of interceptors cannot change after ORB initialisation. The *extensible transport framework* supports developing and integrating new transport protocol implementations to be used by the ORB [10].

CCM extends the basic CORBA object model to support the concept of *CORBA components*, which interact through interfaces and events and execute within run-time execution environments called *containers*. Containers are responsible for providing infrastructure services to a hosted component, such as transactions, persistency, security, events, and lifecycle management. Components can access these services both *explicitly*, through invoking interfaces on the container and receiving invocations on their interfaces, and *implicitly*, through having the container intercept incoming calls on the component and perform pre- and post-processing (e.g. starting transactions). This architecture simplifies component development because it allows developers to concentrate mainly on functional concerns while containers manage extra-functional concerns. The container behaviour is configured declaratively using *component descriptors*, which specify requirements of components on container-provided services. For example, a component descriptor may specify the transaction policies and security rights for a particular operation of the component. Descriptors are set statically, before component deployment.

Evaluation

Support for extension with respect to binding types is unavailable in CORBA. The ORB provides only a remote method invocation binding type. CORBA services such as the event service [11] or the audio/video streams service [12] represent ad hoc attempts to provide additional binding types without systematically addressing the extension requirement. Indeed, the APIs of the different services have little in common, and their implementations rely commonly on distinct infrastructures. This lack of integration imposes complexity on using and implementing binding types in the form of CORBA services. CCM combines remote method invocation with event delivery but still suffers from lack of extensibility. Support for binding adaptation is limited; the only exception is Real-time CORBA which defines policies that control the selection and configuration of protocols on the server and client side of the ORB.

Facilities for resource adaptation are provided by Real-time CORBA for a small set of resource types. Specifically, Real-time CORBA provides control over transport connections, threadpools (i.e. groups of threads for processing requests on the server side of the ORB), and buffer memory associated with threadpools. Moreover, it supports schedulable entities (termed distributable threads) that span node boundaries, and it enables dynamic customisation of their scheduling parameters (e.g. priority or deadline). Real-time CORBA also supports extension with respect to scheduling policies (e.g. fixed priority or earliest deadline first).

The major limitation of CORBA is the lack of flexibility that results from the limited support for extension. Policies and component descriptors support selection between predetermined options. The set of container-provided services in CCM is fixed. Only a small number of platform elements are open to extension—notably, transport

protocols, interceptors, and scheduling policies—and no support is defined for dynamic integration of extension components. Moreover, although large-scale changes for accommodating different environments are performed regularly within OMG (e.g. deriving Real-time and Minimum CORBA from basic CORBA), CORBA defines no systematic approach for performing them.

Ease of customisation is supported by the policy mechanism, which provides consistency for developers. CCM supports higher ease of customisation and use than basic CORBA (i.e. the ORB and common services). The reason is that CCM allows many middleware services to be accessed implicitly and configured declaratively. However, implicitly-accessed services embody pre-packaged, common usage patterns of basic CORBA services, thus exposing less flexibility to application developers.

2.2.2 J2EE and Jini

Here we discuss two middleware platforms from Sun Microsystems: the Java 2 Platform Enterprise Edition (J2EE) and Jini. Both are based on the *Java platform*, which specifies a virtual machine, termed Java virtual machine (JVM), and a set of supporting APIs [13]. *J2EE* extends the Java platform with specifications for developing and deploying multi-tier, enterprise applications [14]. Specifically, J2EE specifications fall into the following categories:

- component technologies supporting different types of application components; namely, application clients, applets, web components, and server-side components, which are known as Enterprise JavaBeans (EJB) components
- communication APIs that support interconnecting application components, including: Java remote method invocation (Java RMI), Java API for XML-based RPC (JAX-RPC), Java IDL, and Java message service (JMS)
- common services that facilitate application development, such as the Java Transaction API (JTA) and the JDBC API for database access

The J2EE customisation facilities are examined next, focusing particularly on the communication APIs and the EJB component technology. *Java RMI* is a core part of the Java platform that enables interoperation between objects that reside in different JVMs. Java RMI is closely integrated with the Java language, which makes it easy to understand and use. The flexibility of RMI is limited to supporting pluggable socket factories that implement various transport protocols. *JMS* is a messaging API that supports both message queuing and publish/subscribe styles of messaging. Its flexibility is limited to supporting selection between two message delivery guarantees: at-most-once and once-and-only-once. *EJB* [15] is a component technology with a container-based architecture similar to that of CCM. In fact, CCM was designed explicitly to provide close correspondence with version 1.1 of EJB in order to facilitate integration between the two technologies. Similarly with CCM, the EJB container provides a fixed set of middleware services with a fixed range of configurations selected statically.

Jini is a Java-based middleware platform that enables services distributed over the network to discover and to interact with each other dynamically [16]. A Jini service is accessed through objects that are downloaded into client machines and communicate

with the remote service using Java RMI or other private mechanisms. In particular, the Jini specification includes:

- basic infrastructure that enables services to lookup, advertise their availability to, and communicate with other services
- a set of APIs for supporting service development; notably, APIs for configuration, leasing, distributed events, and transactions
- a set of standard Jini services; notably, the JavaSpaces service which provides a shared data space for storing and retrieving Java objects over the network

As regards customisation facilities, the basic infrastructure contains an enhancement of Java RMI, called *Jini Extensible Remote Invocation (JERI)*, which provides superior flexibility. In particular, JERI defines a protocol stack architecture and allows selection of the client- and server-side implementations of all layers for a particular remote object. Moreover, it allows clients to control method invocations dynamically using predefined, declarative constraints, such as constraints on the confidentiality and integrity of transmitted data. Another notable customisation facility is the *Jini configuration API* that supports deployment-time customisation and extension of Jini applications. Specifically, the API defines a uniform way of obtaining configuration objects constructed from data in text files or other sources.

Evaluation

Extension with respect to binding types is unsupported in J2EE and Jini. Instead, the platforms specify a set of separate interaction mechanisms with different APIs, such as Java RMI, JAX-RPC, Java IDL, JMS, distributed events, and JavaSpaces. Binding adaptation is supported only by JERI and, to a lesser extent, by JMS. With respect to resource management, both platforms rely solely on Java. The Java platform provides an abstraction of underlying resources (e.g. threads and sockets) but no support for monitoring and controlling the resource usage of Java applications. This limitation is the subject of much current research [17][18][19][20]. Extensibility in resource management policies is similarly lacking.

J2EE and Jini suffer from a lack of flexibility. In particular, both platforms lack support for dynamic changes, and J2EE also lacks support for extension. The dependence on a single programming language impairs further the flexibility and applicability of the platforms. Although languages other than Java can be supported by the JVM, multi-language support was not an explicit goal of the JVM and presents several difficulties [21].

The EJB specification facilitates middleware use and customisation through its support for declarative access to middleware services. Compared to EJB, Jini provides lower ease of use since it exposes a low-level, procedural programming model. Jini facilitates customisation through its consistent configuration model. Finally, the dependence on the Java platform introduces a performance overhead since the JVM mediates all access to the underlying infrastructure.

2.2.3 COM and .Net

In this part we discuss two middleware platforms from Microsoft: the Component Object Model (COM) and the .Net framework. *COM* is a component technology that supports in-memory interoperation between independently developed components, possibly written in different programming languages [22]. Interoperation relies on a binary standard that defines how interfaces are represented in memory and how their operations are invoked dynamically. Over time, the COM technology has grown to include:

- support for invoking methods on COM components that reside in different processes and machines; this support is known as *Distributed COM (DCOM)*
- an extension of COM, known as *COM+*, that provides components with a set of declaratively-configured services, such as transactions, security, events, and synchronisation

The main customisation facility provided by DCOM is *custom marshalling*, which allows objects to select custom proxy implementations to be created on the client side. COM+ has a container-based architecture that employs interception and declarative requirements on container-provided services. Similarly to CCM and EJB, the set of services is fixed and their configuration is static.

The *.Net framework* is the latest, programming language-neutral component technology introduced by Microsoft [23]. It consists of a virtual machine, called the common language runtime (CLR), and a set of supporting APIs. The part of .Net concerned with interacting with remote objects is called the *remoting framework*. This framework provides rich configuration and extension facilities, such as custom proxies, pluggable communication channels (e.g. TCP and HTTP channels) and pluggable formatters (e.g. binary and XML formatters). Moreover, similarly to COM+, the remoting framework defines a container-based architecture for transparently providing services to objects. Specifically, services are provided by *contexts*, corresponding to the container abstraction, which host one or more objects.

The context within which an object resides depends on *context attributes* associated with the object's class; the context attributes express requirements on context-provided services. Importantly, unlike COM+, the set of services is extensible. Custom services are realised as sets of *message sinks* that intercept and manipulate cross-context invocations. Custom services are configured through custom context attributes.

Evaluation

COM and .Net provide no support for extension with respect to binding types. Moreover, binding adaptation is static: neither DCOM nor the remoting framework support customising bindings while they are being used. Support for customisation of resource management is lacking. Specifically, COM defines no resource-related facilities at all, and .Net, similarly to Java, offers no resource adaptation facilities.

Because of the support for extensible container-provided services, .Net provides higher flexibility than similar container-based technologies, such as COM+, CCM and

EJB. However, dynamic flexibility is still lacking in .Net. While one can configure aspects of the remoting framework at various times, the resulting configurations remain normally static. For example, the set of context services provided to an object and their properties cannot be changed after object instantiation. COM provides no support for dynamic changes. Its custom marshalling facility exposes a high degree of extensibility as it enables replacing the entire remoting infrastructure with a custom one. However, support for developing such a custom infrastructure is lacking. For example, one cannot reuse parts of the standard remoting infrastructure, which must be used either completely or not at all. In contrast, the .Net remoting framework provides stronger support for assembling such infrastructures, mainly in the form of pluggable channels and formatters. Developing custom services in .Net is more complex as the framework defines only primitive abstractions, such as message sinks and messages.

Because of the memory and processing overheads introduced by the CLR, the performance of .Net is lower than that of COM. Finally, the applicability of both platforms is affected negatively by their close ties to a single vendor. As regards COM, there are implementations of its core part and DCOM on other operating systems, but COM+ is implemented only on Windows. As regards .Net, Microsoft has made publicly available a specification of (a subset of) the platform, and there are on-going efforts to provide implementations on other operating systems.

2.3 Historical Overview of Predecessors of Aspect-oriented Middleware Technologies

2.3.1 Separation of Concerns

Since the earliest days of computing, developers have strived to create programs in a modular fashion in order to make code more understandable, thus decreasing complexity and easing maintenance and possibilities for evolution. In the 1970s, computer science pioneers Edsger Dijkstra and David Parnas wrote seminal papers on the importance of program structure. In [24], Dijkstra wrote about the importance of *separation of concerns*, or as he states, “to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency”. Parnas, in [25], describes a systematic approach to decomposing a system into modules in order to improve comprehensibility, changeability and promote independent development. Suffice to say, that over three decades later, the principles set forth by Dijkstra and Parnas are still very much at the heart of software engineering.

Modern software development centres around object-oriented programming (OOP). OOP encourages the developer to think in terms of *objects*, and their *behaviour* (methods) and *attributes* (fields). For example a simple ‘engine’ object may have behaviour to start and stop the engine, and attributes such as capacity, temperature and revolutions per minute among others. By thinking of a system in terms of collaborating objects and their operations, provides an effective abstraction for translating requirements to designs and code.

2.3.2 Design Patterns

Over the last decade the use of *design patterns* in software development has become very popular. Design patterns are the reuse of good and well established ideas to

solve a particular problem. The idea for design patterns originates from the architect Christopher Alexander who wrote two books in the 1970s, namely “*The Timeless Way of Building*” [26] and “*A Pattern Language*” [27], which provide ideas and patterns for creating high quality architectural structures. While the concepts were originally developed with buildings in mind, the same principles have been utilised in the development of software architectures and artefacts. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, or as they are more popularly known “The Gang of Four” or GoF, further popularised the use of patterns in software design with their book *Design Patterns – Elements of Reusable Software* in 1995 [28].

2.3.3 Reflection

Reflection [29] is the ability for an executing program to discover information about itself at run time and modify its own behaviour and structure accordingly. A reflective language has the capability to *meta-program*; the ability to write its own programs based upon its own conditions. Many object-oriented languages support reflection to some kind of degree. Smalltalk [30], Python [31] and Ruby [32], in particular, have comprehensive reflection mechanisms, while Java supports it in a less flexible and less dynamic fashion. Reflection has garnered a great deal of interest within the AI community where it presents a natural relationship for neural networks and fuzzy logic.

2.3.4 Component Based Software Engineering

The term ‘software component’ was first used by McIlroy [33] at a NATO conference in 1968. Component-based software engineering (CBSE) [34] refers to the development of software systems from reusable modules known as *components*. With CBSE, software applications can be composed together by simply bolting together ready made components thus greatly simplifying software development and reducing costs and time to market. Szyperski defines a component as “... a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [35].

Interface description languages (IDL) are used to describe the component’s interface in terms of the services it *provides* and *requires*, although the original CORBA IDL did not describe the requires services. Architectural description languages (ADL) work at a higher level of abstraction and are used to describe the software architecture by declaring components, connectors (interactions between components) and configuration details (topology). Some examples of ADLs are ACME [36], Darwin [37], Wright [38] and Rapide [39].

2.3.5 Object Oriented Frameworks

Frameworks have emerged as a promising approach to developing quality software within time and cost constraints. Johnson and Foote describe a framework as “... a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes.” [40] An alternative way to think of a framework is to view it as a semi-complete application which can be

easily changed by binding new behaviour at predetermined variation points or “hot spots” in the implementation.

Fayad and Schmidt, in [41], describe the four primary benefits of frameworks as being “...modularity, reusability, extensibility and inversion of control they provide to developers.”

- *Modularity* helps to encapsulate design implementation and the impact of change improving system understanding and the activities of maintenance and evolution.
- *Reusability* is enabled via generic components which allow components to be used in different contexts.
- *Extensibility* is enhanced with the provision of explicitly defined hooks in the framework.
- *Inversion of control* enables the developer to concentrate on application specific needs rather than worry about issues such as program structure and control flow as these will be already encapsulated within the framework.

2.3.6 Compositional Adaptation

The emergence of mobile, *ubiquitous* [42] and *autonomic* [43] computing has led to an interest in *compositional adaptation* [44]. Compositional adaptation allows software to dynamically adapt new behaviour, in a transparent way, based upon changes in its environment. For example, a running application could adapt new algorithms for concerns, which were not known at development time.

2.4 Research Middleware Platforms

Here we outline and evaluate a selection of chosen research work that addresses the problem of customisability in middleware.

2.4.1 FlexiNet

FlexiNet [45] is a Java middleware platform developed by APM Ltd. that focuses on remote method invocation bindings. It is structured as a white-box object oriented framework that supports assembling protocol stacks, which are responsible for all aspects of invocation processing. This includes not only basic features such as implementing a remote procedure call protocol (e.g. IIOP) but also higher level features such supporting replication or encryption. Each layer of the protocol stack manipulates invocations represented in a generic form before they are ultimately invoked on destination objects.

Assembling a protocol stack involves managing interdependencies between layers and helper objects. To support this task, FlexiNet provides the *blueprints* mechanism, which supports specifying arbitrary object graphs and then coordinating their construction. Specifically, graphs are specified using constraints and suggestions for the possible objects to be placed at graph nodes (e.g. possible objects that implement transport protocols). If a subsequent resolution step completes successfully, each node denotes a single object and the graph can be constructed and initialised; otherwise, failure is reported.

FlexiNet enables applications to control both the protocol stack and associated stack-specific QoS properties that will be used for binding to a specific object. Moreover, it provides basic support for dynamic binding adaptation by defining a generic configuration interface on client-side proxies. *FlexiBind* extends this basic support by enabling pluggable *policies* that control the configuration of protocol stacks and pluggable *meta-policies* that control the selection of policies [46].

Evaluation

FlexiNet provides no support for adding new binding types beyond remote method invocation. It supports binding adaptation mainly at binding establishment time. The minimal support for dynamic adaptation is enhanced by FlexiBind. Basic support for controlling resources is provided through the definition of a resource pool abstraction associated with pluggable allocation policies. Resource pools manage threads, buffers, and sessions and are assigned to protocol stack layers.

The overall flexibility is restricted by the lack of support for large-scale changes. With respect to ease of customisation, building protocol stacks is complex since the framework leaves a large amount of freedom to developers. The blueprints mechanism facilitates modifying existing stacks incrementally, without detailed knowledge of the stack structure. Moreover, it supports reliability by enabling the association of constraints with existing stacks. However, the mechanism does not facilitate building new stacks, and the resolution process is inflexible and costly. Finally, FlexiNet is a Java-only solution, which restricts both its applicability and performance.

2.4.2 Jonathan

Jonathan [47][48] is a Java middleware platform developed originally at the research labs of France Télécom. It is designed as a white-box object oriented framework that supports composing different middleware systems implementing different *personalities* (i.e. sets of APIs). Jonathan also includes a library of middleware building blocks that conform to the framework and populate the implementations of two personalities: a CORBA personality and a Java RMI personality. Specifically, the current version of Jonathan (3.0) is organised into four parts: the binding framework, the communication framework, the resources framework, and the configuration framework. The *binding framework* supports the development and introduction of new binding types in the form of pluggable binders. *Binders* are responsible for managing references to objects and providing access to referenced objects. Implemented binders include an IIOP binder that implements remote method invocation using CORBA IIOP and an event channel binder that implements a publish/subscribe interaction style. The *communication framework* supports composing protocol graphs from reusable protocol implementations. Implemented protocols include the IP Multicast protocol, Real Time Protocol (RTP), and CORBA GIOP. The *resources framework* provides abstractions for managing buffers, threads, and network connections and supports extension with respect to associated management policies. Finally, the *configuration framework* provides a generic way to establish the initial configuration of a system using an XML-based structural

description. The framework is used to assemble a middleware system at deployment time by instantiating and interconnecting various Jonathan classes, such as binders, protocols, and resource management policies.

Evaluation

The distinguishing feature of Jonathan is its support for extension with respect to binding types. This feature enables the platform to support high-level interaction mechanisms, thus enhancing its ease of use. Jonathan supports binding adaptation but only at binding establishment time. Customisation with respect to resource management is supported through the resource framework.

Regarding flexibility, Jonathan lacks support for dynamic customisation. It does support large-scale variations in the form of different personalities, but the associated flexibility is limited. The reason is that Jonathan personalities mainly capture variations in surface features of the middleware system, such as marshalling, rather than variations in the structure and behaviour of the frameworks. The flexibility embodied in the binding framework is limited by the emphasis on *first-party binding*, which means that the party that initiates and controls binding establishment is one of the communicating participants. No explicit support for the general case, called third-party binding, is provided.

Static customisation is facilitated by structuring Jonathan into separate frameworks and by the configuration framework. No special support for consistence management is provided. Finally, the applicability and performance of Jonathan are restricted because of its dependency on Java.

2.4.3 OpenORB

OpenORB (or more precisely, OpenORB version 1) is the first generation of reflective middleware developed at Lancaster University [49][50][51]. In essence, OpenORB is a component technology that supports distribution and provides sophisticated reflective facilities. OpenORB has been implemented by a set of prototypes written in the Python programming language [52][53][54]. Heavily influenced by RM-ODP, OpenORB defines three fundamental concepts: components, interfaces and bindings. *Components* are run-time entities that expose one or more access points, that is, *interfaces*. Interfaces fall into three categories—i.e. operational, stream, and signal interfaces—that support corresponding interaction styles—i.e. operation invocation, continuous media interaction, and atomic one-way interaction. Interfaces are connected through *bindings*, which fall into two categories: *local bindings*, i.e. primitive constructs that directly connect interfaces in the same address space, and *explicit bindings*, i.e. first-class objects that connect interfaces potentially located in different address spaces. Components and explicit bindings can be composite; that is, they may have an internal structure comprising components and explicit bindings connected by local bindings.

The reflective facilities in OpenORB support inspection and dynamic adaptation of multiple aspects of components and bindings. Specifically, the facilities are structured into four *meta-models*, namely, the interface, architecture, interception, and resources

meta-model. The *interface meta-model* provides access to the external view of components and bindings; it enables enumeration of provided interfaces and discovery of interface definitions. The *interception meta-model* enables the dynamic attachment of interceptors to interfaces, which allows the insertion of pre- and post-processing functionality. The *architecture meta-model* provides access to the internal structure of components and bindings represented as an object graph; it provides operations to retrieve, insert, remove and replace components and explicit bindings, as well as operations to manipulate the local bindings connecting them. The *resources meta-model* provides access to underlying resources and resource management. Specifically, the meta-model captures diverse types of resources at different levels of abstraction (e.g. buffers, user-level threads, and kernel-level threads), and provides control over the distribution of resources among *tasks*, defined as units of resource allocation. Tasks are invocation sequences that can span multiple components distributed over different address spaces.

Evaluation

OpenORB supports extension with respect to binding types. This is achieved by adding new binding factories, that is, special components responsible for binding establishment. Binding adaptation is also supported since established bindings are represented as objects (explicit bindings), which can be manipulated through the meta-models. Customisation with respect to resource management is enabled by the resources meta-model.

OpenORB supports a high degree of flexibility as virtually every aspect of a middleware system is available for dynamic customisation and extension. One aspect in which flexibility is weaker is the range of binding types that can be implemented as binding factories. The first-class support for the three interaction styles in the component model makes it difficult to implement other unrelated styles. For example, adding support for group communications to an OpenORB prototype required introducing a new API and modifying the implementation of interface references.

The main disadvantage of OpenORB is that a large amount of effort is required for middleware customisation. Indeed, it is difficult to develop new middleware functionality, such as binding factories employing specific protocols or scheduling policies, since OpenORB provides only generic mechanisms for assembling components and bindings. It is also difficult and error-prone to modify existing middleware functionality, such as explicit bindings, since meta-models expose low-level primitives to modifiers (e.g. component replacement). On the other hand, meta-models act as uniform and well-structured customisation facilities, thus helping reduce the complexity imposed to modifiers.

Performing unrestricted changes using meta-models can easily compromise the consistency of a middleware system. OpenORB provides no effective support for consistency management. Related work in Lancaster addresses this issue by enhancing the architecture meta-model to maintain architectural style rules and to ensure that the rules are preserved before any changes are committed [55]. Apart from reducing flexibility, this enhancement comes at the cost of ease of customisation and performance.

Finally, the reflective facilities in OpenORB impose storage and processing costs in terms of maintaining and keeping consistent self-descriptive information, such as object graphs. Importantly, since the reflective facilities are an inseparable part of the component model (e.g. all components have architecture meta-models), this overhead cannot be avoided or scaled down depending on application requirements.

2.4.4 2K, dynamicTAO, and UIC

2K is a distributed operating system for dynamic heterogeneous environments developed at the University of Illinois at Urbana-Champaign [56]. 2K adopts a layered architecture in which applications build on a middleware layer that builds on a kernel layer. The kernel layer comprises a conventional operating system or the 2K microkernel. The middleware layer comprises a set of distributed operating system services running on top of reflective CORBA-compliant ORBs. These services include standard CORBA services (e.g. naming, trading, and security service) as well as services for automatic configuration, resource management, and code distribution. The reflective ORBs used in 2K are dynamicTAO and the Universally Interoperable Core. 2K customisation facilities are examined next focusing on the automatic configuration service and the reflective ORBs.

The *automatic configuration service* supports deploying a software component by means of meta-information about its dependencies, called *prerequisites* [57]. Prerequisites describe: (1) the type and capacity of hardware resources that the component requires, and (2) other supporting components on which the component depends. The first kind of prerequisites is passed to the 2K resource management service in order to perform resource allocation. The second kind is used to load the supporting components, if necessary, as well as to create run-time representations of the dependencies. Run-time dependencies are maintained by entities known as *component configurators*. Specifically, a configurator holds the dependencies between a certain component and other components and exposes standard operations to manipulate these dependencies dynamically and to receive events from these components. The implementation of a configurator can be customised to exploit application-specific knowledge; for example, a configurator can disallow removing a dependency on a component under application-specific conditions.

DynamicTAO [58] is a reflective ORB built as an extension of TAO [59], a modular and configurable middleware platform based on design patterns and developed at the University of Washington. TAO uses the *strategy* design pattern to encapsulate different aspects of the middleware implementation. DynamicTAO uses customised component configurators to enable dynamic reconfiguration of strategies while preserving consistency. The *Universally Interoperable Core (UIC)* is a reflective ORB targeting environments with limited resources, such as handheld devices [60]. UIC defines a skeleton of abstract components that encapsulate standard functional aspects of ORBs (e.g. marshalling strategies and concurrency policies), and it can be specialised to form different *personalities* (e.g. CORBA client-side personality or Java RMI personality). Specialisation in UIC involves developing concrete components that conform to the abstract components and inserting them into the skeleton structure. Specialisation may also include changing the skeleton itself in order to perform larger-scale changes. Specialisation may happen at compile-time, by

statically linking components to form a monolithic personality, or at run-time, by dynamically loading and unloading components to upgrade personalities. As in dynamicTAO, dynamic changes are managed using a component configurator.

Evaluation

With respect to binding management and ease of use, 2K provides no advantage over CORBA; it supports only remote method invocation bindings, and binding adaptation is limited. The 2K resource management service supports resource control, but this is currently restricted to a single resource type, i.e. CPU time, on a coarse-grained basis, i.e. per-process. DynamicTAO inherits basic support for resource management from TAO, which exports OS-managed resources through common interfaces.

DynamicTAO and UIC provide dynamic flexibility by means of configurators, but this flexibility is restricted to replacing shared, coarse-grained middleware components (e.g. the ORB-wide concurrency strategy). Large-scale customisation is supported in UIC, but not in dynamicTAO. Moreover, the flexibility and applicability of both platforms are affected negatively by their dependence on single-language components (i.e. C++ components). Regarding ease of customisation, developing new middleware functionality is guided by applications of the strategy and other patterns in dynamicTAO and the skeleton structure in UIC. The initial assembly of components is facilitated by the automatic configuration service. Dynamic reconfiguration is facilitated by the uniform interface exposed by configurators and the clear separation between using and reconfiguring a middleware system. In addition, the configurator interface is well-suited for performing the intended changes (i.e. replacing shared, coarse-grained components), thus imposing a low workload on modifiers. Importantly, both this low workload on modifiers and the coarse-grained flexibility mentioned earlier are characteristics of dynamicTAO and UIC, not of the general configurator mechanism. In principle, employing a large number of configurators in an architecture would enable fine-grained flexibility, but at the cost of simplicity and performance.

Reliability is supported by allowing customised implementations of component configurators; these configurators can exploit context-specific knowledge in order to perform reconfiguration without compromising the consistency of the middleware system. For example, when replacing the thread-pool concurrency strategy, the TAO-specific configurator takes special care to destroy all threads contained in the pool before unloading the strategy code. Finally, maintaining dependencies and loading and unloading components impose storage and processing costs. In particular, processing costs are only incurred during reconfiguration activities. The overall overhead depends on the number of configurators, which is small in dynamicTAO and UIC.

2.4.5 Customisation of ORBs by Jørgensen et al.

Researchers at the University of Southern Denmark and K.U. Leuven have developed a platform that supports customising ORB implementations to non-functional application requirements [61][62]. ORB implementations are based on component-based ORB architectures that are specific to application domains. Customisation is

realised through automatic, dynamic selection between alternative component implementations. The platform has been applied in the domain of robotic control applications using an ORB architecture based on the JavaBeans component model [63]; this architecture supports customisation according to temporal non-functional requirements, such as timing constraints on invocations.

In more detail, an *ORB architecture* is defined as a set of component types linked by connectors. *Component types* define provided and required interfaces, and *connectors* connect required interfaces to provided interfaces. A *component instance* provides an implementation for a specific component type, and a component type may be implemented by more than one component instance. Each component instance is associated with a set of *component descriptors* that describe *provided* service levels for non-functional requirements supported by this instance. Similarly, each application method is associated with a set of *application-specific policies* that describe *expected* service levels for non-functional requirements (e.g. expected deadline or period associated with invocations). Both descriptors and policies are expressed using the same vocabulary.

Customising the ORB consists in selecting a single component instance per component type to handle a particular method invocation. Specifically, method invocations and their associated policies are reified as objects that flow through the system from one type to another. At each step, the system inspects and matches the policies of the invocation with the descriptors of the alternative instances belonging to the type. The best matching instance is then selected for handling the reified invocation. An additional form of customisation involves dynamically injecting *wrappers* that adapt the behaviour of specific component types. Wrappers are intended to support the dynamic introduction of new types; for example, a wrapper of the transport type can introduce an encryption type in the ORB architecture.

Evaluation

The platform concentrates on remote method invocation bindings, and binding adaptation is static since policies are associated with method definitions. However, since policies are interpreted at the time of each invocation, the platform can, in principle, accommodate dynamic policy changes. The platform does not address resource adaptation. In the implemented ORB architecture, customisation of resource management is limited to selecting among alternative scheduler instances (e.g. a real-time and a FIFO scheduler instance).

The degree of dynamic flexibility embodied in policy-driven customisation is restricted to switching between instances in a fixed run-time structure with fixed connections. The mechanism for injecting wrappers increases flexibility but only to a limited extent since wrappers must comply with the interfaces of wrapped components and can only add pre- and post-processing functionality. The platform allows large-scale changes through defining domain-specific ORB architectures.

The main advantage of the platform is the ease of customising the ORB implementation. For application developers, customisation involves specifying service expectations for non-functional requirements in a declarative language. For middleware developers, customisation involves implementing components that conform to certain types and specifying their service provisions. Using the wrapper

injection mechanism is far more difficult and error-prone. Defining domain-specific architectures is allowed but not particularly supported. Regarding consistency of the running system, this is preserved as long as component instances conform to their expected types. The platform is easy to use because of the declarative access to middleware functionality. Finally, the platform introduces a potentially high performance overhead since policies are interpreted multiple times per method invocation.

2.4.6 Multe-Orb

The goal of the MULTE (Multimedia Middleware for Low-Latency High-Throughput Environments) [69] is to design flexible and adaptable middleware systems that support a broad range of QoS requirements, e.g. low-latency, high throughput and controlled delay jitter. Multe-Orb is the first prototype that is based on a CORBA 2.0 implementation (COOL 4.1) and the flexible protocol system Da CaPo (Dynamic Configuration of Protocols). Multe-Orb allows application specified QoS related parameters to be instantiated as a set of protocol modules that jointly achieve the required QoS specification.

Multe-Orb aims at integrating an end-to-end QoS solution with a standardised ORB in the least intrusive way. At the application side, Multe-Orb supports a QoS specification at the CORBA binding and method invocation level. At the protocol layer, QoS is agreed by using a simple parameter negotiation protocol between client and server. The Multe-Orb prototype focuses on providing QoS support only for CORBA request-reply invocations.

Because of limitations in the TCP/IP protocols with regards to parameterising QoS requirements, Multe-ORB integrated the Da CaPo protocol architecture into the underlying COOL ORB. Da CaPo encapsulates protocol tasks like error detection, acknowledgments, etc. into different protocol modules, allowing the arbitrary combination of such modules in a so-called module graph, which in turn supports different QoS. Supporting QoS at the protocol layer requires some minor but necessary changes at the GIOP protocol headers.

Evaluation

The dependence of Multe-Orb on the Da CaPo protocol architecture enables run-time binding adaptation and potential extensibility of binding types. However, the reliance on the CORBA standard and its orientation towards method invocation semantics makes extensions to binding types quite difficult. The current version of Multe-Orb supports only request-reply communication semantics. Run time adaption is supported per method invocation with support for monitoring of network and application conditions that can trigger protocol reconfiguration of protocol modules.

Facilities for resource adaptation are provided by the monitoring component in Da CaPo. The monitoring component controls the availability of end-system resources and the connection properties. Customisation of these facilities can be carried out by the application, allowing dynamic adaptation to different levels of QoS. There is no provision in Multe-Orb for a policy framework with which to extend resource

management facilities. As such, connection properties are described in an attribute-value syntax and are being negotiated with peers with a simple unilateral negotiation protocol. This reduces the flexibility of resource management tasks. The overall flexibility is restricted by the dependence on the CORBA architecture. There does not seem to be any provision for consistency management other than what is performed during composition of protocol modules from the application QoS specification.

2.4.7 K-Components

The K-Component architecture meta-model [64] is designed to build dynamically adaptable software architectures. The architecture configuration is stored as a typed connected graph, where the vertices are interfaces that are labelled with the component instances that implement them. Interfaces are connected by directed edges labelled with connector properties, which represent the reconfigurable properties of the connector such as the ability to change its communication protocol, etc.. The entry point in the program is represented as the root of the vertex. The graph is automatically generated from the component definitions and the actual implementation code. It is stored and managed by a meta-level component called the configuration manager. Dynamic reconfiguration is achieved using reflective code called *adaptation contracts*. These adaptation contracts specify conditional transformations based on architectural constraints. They are implemented as meta-level objects that can be loaded and unloaded at run-time using the configuration manager. The integrity of the system is maintained by the configuration rules specified on the edges of the graph and by a reconfiguration protocol that ensures that vertices involved in the reconfiguration are in a safe state. Further adaptation contracts also take care of the management of incoming and outgoing dependencies of the system. The adaptation code is kept separate from the computational code by using a special Adaptation Contract Description Language (ACDL) to specify this code. The K-Components system implements components with an architecture meta-model, and adaptations contracts to support reconfiguration.

Evaluation

The K-Component model provides no support for adding new binding types, and its basis on the CORBA model restricts it to the RPC communication paradigm. Communication through an event model is reserved for the adaptation contracts. Also, as the only supported binding is RPC, contextual adaptation is beyond the scope of K-Components.

K-Components inherit the platform abstraction of the underlying CORBA implementation. Therefore, the underlying middleware has to be instrumented to provide adaptation events to adaptation contracts in order to adapt to changes in middleware. The ACDL provides a declarative policy language and is complemented by a mechanism for static and dynamic adaptation between connected components. The contracts can monitor connectors that may cause adaptation actions to be executed, e.g. reconfigure a connector to use some other version of a service.

K-Components provides explicit support for the dynamic adaptation of its architecture meta model, and programmer-supplied component-specific adaptation actions but offers limited support for extensions and large-scale customisations. Policies for

adapting a K-Component can be specified as rule-based policies, event-condition-action policies, reinforcement learning policies and collaborative reinforcement learning policies.

Ease of customisation in K-Components is restricted by the underlying platform, while provision for a form of reliability is provided by an RPC-consistency protocol that ensures that components can only be adapted when there is no ongoing computation or outstanding communication with other components. This guarantees the integrity of adapting components.

2.4.8 Quartz

Quartz [65] defines an architecture that provides support for quality of service (QoS) specification and enforcement in heterogeneous distributed computing systems. The Quartz QoS architecture has been designed to overcome various limitations of previous QoS architectures that have constrained their use in heterogeneous systems. These limitations include dependencies on specific platforms and the fact that their functionality is often limited by design to one particular area of application. Quartz is able to accommodate differences among diverse computing platforms and areas of application by adopting a flexible and extensible platform-independent design, which allows its internal components to be rearranged dynamically in order to adapt the architecture to the surrounding environment. Other restrictions, such as the lack of flexibility and expressiveness in the specification of QoS requirements and limited support for resource adaptation, are also addressed by Quartz.

Applications requiring QoS enforcement use the mechanisms provided by Quartz to specify their requirements. In order to enforce the required QoS, Quartz employs the resource reservation protocols available in the target network and operating system. Quartz defines two main levels of abstraction for QoS specification, the application level and the system level. A Translation Unit that is part of the middleware architecture is responsible for translating the application defined QoS specification to a set of parameters corresponding to the available protocol of the hosting platform. The central component in Quartz is the QoS agent, that is composed of the Translation Unit and multiple System Agents associated with the reservation protocols responsible for administering the use of the available resources.

Evaluation

Quartz provides no support for extending binding types. As it is not a generic middleware architecture, but rather a QoS architecture, extending binding types is outside the scope of this work. Quartz provides extensive infrastructure for dealing with resource adaptation and reservation. Adaptation rules can be specified at the system and the application levels. At the system level, the System Agent is used to adapt to environmental condition including network or host resources. The System Agent is also responsible for monitoring of the resources that are occupied by the reservation protocol it corresponds to. At the application layer, Quartz employs a declarative attribute-value based syntax for resource reservation. Application adaptation is provided in the form of call-backs from the middleware. Applications

are notified only in case the middleware is unable to provide the resources originally requested.

One of the stated goals of Quartz was to provide a flexible and adaptable architecture. As such, Quartz is platform independent allowing both static and dynamic customisation of applications running on top of it as described above. Additionally, Quartz supports extensibility by wrapping resource reservation protocols inside its System Agent module. This has the effect of increasing the portability of applications across different platforms, and makes it easier to extend the architecture in order to support new resource reservation protocols. However, Quartz offers limited support for system-wide customisation, and only provides system agents with a set of classes from which they can inherit common behaviour. Reliability is not addressed in the version of Quartz described in [65].

2.4.9 QoS-enabled middleware

Much research activity has concentrated on integrating QoS support in middleware platforms over the last years. For example, the QuO framework [66] defines languages and mechanisms that support developing CORBA applications with QoS requirements. CIAO [67] is a CCM implementation that supports composing QoS provisioning policies with application components statically. DotQoS [68] is a QoS management framework that supports static and dynamic QoS provisioning for multiple QoS categories on top of .Net.

All such QoS-enabled platforms promote a separation of concerns between application development, QoS provisioning, and infrastructure development. QoS provisioning, in particular, must build on customisation facilities in the infrastructure, such as facilities for invocation interception, or resource monitoring and control. As demonstrated by DotQoS, a flexible, modern middleware infrastructure, such as .Net remoting, facilitates greatly the development of QoS-enabled platforms. However, since the flexibility of such QoS-enabled platforms is bound by that of the underlying middleware infrastructure, they are not examined further.

2.5 Evaluation Overview

This section presents first a quantitative overview of the evaluation and then a more detailed platform comparison and general discussion.

2.5.1 Quantitative view

Table 1 provides a quantitative, simplified view of the preceding evaluation of platforms. In this table, functional requirements can be supported, not supported, or partially supported. The extent to which quality requirements are achieved can be very low, limited, sufficient, or very high. To quantify this extent, the following guidelines are followed:

- Flexibility is sufficient when a platform satisfies adequately all three constraints—i.e. support for static and dynamic customisation, extension, and large-scale customisation.

- Ease of customisation is sufficient when a platform supports developing new middleware functionality and modifying existing functionality through easy to use, well-factored frameworks and interfaces.
- Reliability is sufficient when a platform provides support that is more effective than simply checking type conformance or defining design rules and relying on modifiers to obey them.
- Ease of use is sufficient when a platform provides either declarative access to middleware services or high-level binding types.
- Performance is sufficient when a platform has no incidental dependencies on inefficient underlying technologies, and the overhead of its customisation facilities is low or can be flexibly adjusted.

For brevity, the table lists the best measure applicable to any of the flavours of composite platforms (i.e. CORBA, J2EE, COM, .Net). For example, CORBA is considered to support (partially) resource adaptation even though this capability is specific to Real-time CORBA and is unavailable, for example, for developing CORBA components.

TABLE 1. Middleware platform comparison

	Extensibility	Binding Adaptation	Resource Adaptation	Extension with Resource Mgt. Policies	Flexibility	Ease of Customisation	Reliability	Ease of Use	Performance
CORBA	✗	✓	±	±	▽	▲	▲	●	▲
J2EE	✗	±	✗	✗	▽	▲	▲	●	○
Jini	✗	✓	✗	✗	▽	○	●	○	○
COM	✗	±	✗	✗	▽	▲	▲	●	▲
.Net	✗	±	✗	✗	○	○	○	●	○
Flexi-Net	✗	✓	✓	✓	○	○	●	○	○
Jonathan	✓	±	✓	✓	○	○	○	●	○
OpenORB	✓	✓	✓	✓	▲	▽	▽	●	▽
Dynamic TAO	✗	±	✓	✓	○	●	●	○	●
UIC	✗	±	✗	✗	●	●	●	○	●
Jørgensen et al.	✗	±	✗	±	○	▲	●	●	▽
K-Components	✗	✗	✗	±	○	▽	▲	○	●
Multe-Orb	✓	±	✓	±	○	○	▽	●	●
Quartz	✓	✗	✗	±	○	●	▽	●	▲

✓	supported	▲	very high
✗	not supported	●	sufficient
±	partially supported	○	limited
		▽	very low

2.5.2 Discussion

This section compares the extent to which requirements are satisfied by the surveyed platforms and discusses different approaches used in satisfying them. Moreover, it identifies the overall shortcomings of the surveyed work.

Extension with multiple binding types is not well-supported by the surveyed platforms. Only Jonathan, OpenORB, Multe-Orb and Quartz address this requirement, and all impose restrictions on the range of supported binding types. Customisation of resource management is supported by several research platforms (e.g. OpenORB, FlexiNet, Jonathan, dynamicTAO). Among these, OpenORB provides higher-level support for resource adaptation, mainly owing to the distributed task abstraction, but it is highly prescriptive, which limits its applicability. Conversely, the remaining platforms provide only basic support for resource management without an explicit treatment of resource adaptation.

The highest degree of flexibility is supported by OpenORB, mainly because of its pervasive use of reflection. Other research platforms lack adequate support for either dynamic changes (e.g. Jonathan and Jørgensen platform) or large-scale changes (e.g. dynamicTAO and FlexiNet). Only UIC supports both these capabilities adequately. Even when the requirement for large-scale changes is addressed (notably, in UIC and the Jørgensen platform), the proposed solution consists only in proposing domain-specific component architectures. Little or no support is provided for defining such architectures.

Commercial platforms still suffer from limited flexibility despite the trend towards supporting customisation. They provide little or no support for dynamic and large-scale changes, and their extensibility is restricted to a narrow range of platform elements. Flexibility is particularly low in container-based component technologies since they support a predefined set of services and parametric configuration. A notable exception is .Net with its extensible container-provided services.

Strong support for ease of customisation is provided by platforms that support configuring middleware functionality through declarative interfaces (e.g. container-based technologies, and the Jørgensen platform) or easy to use, procedural interfaces (e.g. dynamicTAO/UIC). Weaker support is provided by platforms that expose unstructured, complex interfaces (e.g. FlexiNet and Jonathan) or interfaces with low-level primitives (e.g. OpenORB). Developing new middleware functionality is supported in almost all platforms through frameworks, either object-oriented (e.g. .Net, FlexiNet, Jonathan) or component-based (e.g. Jørgensen platform). More prescriptive frameworks (e.g. dynamicTAO, Jørgensen platform) provide stronger support than less prescriptive (e.g. .Net. extensible services). One limitation of object-oriented frameworks is that they tend to contain dependencies on implementations (i.e. classes) rather than only interfaces, which complicates framework evolution. A consequence is that it is difficult to perform large-scale changes in platforms based on object-oriented frameworks.

Platforms that support reliability without restricting flexibility to parametric changes are FlexiNet and dynamicTAO/UIC. FlexiNet uses the blueprints mechanism that permits the association of constraints with protocol stacks. DynamicTAO and UIC use

customised component configurators that exploit application-specific knowledge to perform reconfiguration without compromising consistency. Ease of use is supported through declarative access to middleware services (e.g. container-based technologies and Jørgensen platform) or high-level binding types (e.g. Jonathan and OpenORB).

Finally, the only platforms that support performance without overly impairing flexibility are dynamicTAO and UIC owing to their C++-based implementation and their low customisation overhead. Remaining flexible platforms depend on virtual machines (i.e. JVM or CLR) or incur significant performance penalties (e.g. OpenORB and Jørgensen platform).

In summary, the evaluation with respect to customisability, ease of use and performance reveals several shortcomings in the surveyed middleware platforms and thus opportunities for further research. The fundamental shortcoming is that no platform satisfies *all* requirements in a *balanced* way. Other shortcomings include: (1) extensibility with respect to binding types is limited, (2) large-scale customisation remains largely unaddressed, and (3) resource adaptation approaches fail to balance usability with flexibility.

2.6 Summary

This section has presented an overview of a number of middleware platforms and their evaluation with respect to the requirements identified. Specifically, the section has covered both commercial platforms and a selection of research platforms. The main outcome of the evaluation is that no current middleware platform addresses all requirements in an effective manner. Moreover, unsatisfactory support is provided in the areas of binding management, resource adaptation, and large-scale customisation.

3. Overview of Middleware Technologies providing an Aspect Oriented Programming Model

3.1 Introduction

An important challenge in the development of distributed applications, using the middleware platforms described in section 2 of this survey, is to achieve an accurate modularisation of applications into software entities (e.g. objects, components, agents, etc.) that are as independent as possible. The main benefits that are expected to be obtained are better (re)use of software entities in different contexts and the reduction of development time, cost, and effort, while improving the flexibility, reliability, and maintainability of the final applications.

However, the decomposition of the functionality of applications in independent software entities is not a trivial task. One of the main reasons is that software entities often need to use some common services, such as security, persistence, transactions, etc., which have nothing to do with the core functionality of these entities, and in consequence, should live outside their definition and implementation.

Recently, several of the middleware platforms described in section 2 have evolved to face the challenge of achieving the definition and the use of common services outside component implementation -- i.e. consider common platform services as crosscutting concerns. However, these platforms have not yet found a proper solution to this tangled code problem, as they only separate a fixed number of commonly used services in distributed systems.

An initial solution was the CORBA interceptors. An interceptor is a hook in the CORBA ORB that is interposed on the invocation and response paths between a client and a target object to invoke any ORB service. However, there are some limitations when trying to use interceptors as aspects. The most important limitations are that it is not possible to forward the intercepted message to a different target and that some implementations do not allow modifying the intercepted message, which makes it impossible to implement many of the essential aspects that are required in a distributed platform, such as, encryption aspects.

Later, component platforms, such as CCM/CORBA and EJB/J2EE, provided new solutions to the same problem by defining the container programming model. Containers encapsulate the component implementation and separately supply a runtime environment that provides access to platform common services. The main advantage of the container approach is that software developers do not have to include code for accessing these services inside the component implementation. However, the use of the container programming model is restricted to the list of services offered by the platform vendor. Thus, it cannot be applied to model, as separate services, any other properties (e.g. fault-tolerance, synchronisation, and domain-specific properties) that may be also intermingled with the code of the core components.

Therefore, in order to incorporate the concept of an aspect, middleware platforms should offer additional mechanisms to separate any kind of crosscutting concern or replicated code. In this section, we describe middleware platforms that provide an

aspect-oriented programming model that allows developers to define the software architecture of their applications in terms of objects or components and aspects. In these platforms aspects are treated as application-level entities. This means that although the platform provider may offer a set of pre-defined aspects to be reused by application developers, they will also be able to separate other kinds of crosscutting properties that were detected in their applications, for instance domain specific crosscutting properties.

3.2 Comparison Criteria

Section 2.1 of this document introduced *customisability* and *usefulness* as the main requirements for middleware. Each of them were then decomposed into a set of criteria (see figure 1) that were used to evaluate the several platforms described in section 2. These criteria are also applicable to the middleware platforms in this section. Specifically, we are going to focus on the *flexibility*, *reliability* and *performance* provided by middleware platforms, considering these criteria as defined in section 2.1 of this document.

In addition, there are other comparison criteria that are worthy of consideration when comparing middleware technologies that offer an aspect-oriented programming model to software developers. These criteria will allow the comparison of middleware technologies according to features that are specific to aspect-oriented software development. Therefore, they will be useful for software developers to help them to decide which middleware technology fits better with their requirements and necessities for a specific project. A list of these criteria and their description is provided below.

1. *Aspect-Oriented Programming model*: The aspect-oriented programming models offered by middleware platforms in this section may be an extension of existing object-oriented or component-based middleware platforms, such as CORBA, CCM/CORBA, EJB/J2EE, etc. or may be a new aspect-oriented programming model defined from scratch.
2. *Primary entities supported*: A specific middleware platform can provide separation of aspects in applications that are object-oriented, component-oriented, agent-oriented, etc. This is important information that software developers need to know in order to decide which middleware platform to use.
3. *Static versus dynamic weaving model*: Aspects' advice and the core application have to be woven to compose the final application. At this point systems can be classified according to their weaving model, where weaving can occur at the time of compilation, at the time of deployment, at loading or at runtime.
4. *Invasive versus non-invasive joinpoint model*: If the kinds of joinpoints defined by an aspect-oriented approach allow the interception of the internal behaviour (e.g. private methods, field access, etc.) of the core entities (e.g. object, component, etc.) in the application, it is said that the weaving model is *invasive*. On the other hand, if aspects can only intercept those points that core entities expose through their public interfaces, the weaving model is said to be *non-invasive*. This is especially relevant in the context of component-based software development.

Considering components as black-box entities, aspects should not intercept points that are part of the internal behaviour of a component. Instead, they should only intercept the interactions among components.

5. *Aspect reusability*: The reusability of aspects in different contexts is a desirable feature of any aspect-oriented technologies. This is especially relevant in middleware platforms in order to reduce the time and cost of final applications developed on top of the platform. Therefore, it will be important to evaluate the middleware platform for AOSD according to whether they treat aspects as context-independent and reusable entities, or not. This means that in addition to the aspect behaviour or advice, aspects may or may not hard-code the aspect pointcuts. The inclusion of aspect pointcuts together with the aspect advice prevents aspects from being reused in different contexts.
6. *Application Extensibility/Adaptability*: The software entities that are part of a final application and the relationships among them may be extended and adapted both at design and at runtime. On the one hand, dynamic weaving supports adding, removing or updating the entities in an application during its execution, resulting in more extensible and adaptable applications. On the other hand, the description of the aspects' pointcuts using a declarative language, instead of programming them, makes it easier to adapt the relationships among aspects and the core entities of the application both at design and at runtime.

3.3 AO4BPEL¹

AO4BPEL [70][71][72][73] is an aspect-oriented extension to BPEL4WS (Business Process Execution Language for Web Services) [133] that allows for more modular and dynamically adaptable web service compositions. AO4BPEL has been developed by the Software Technology Group at Darmstadt University of Technology.

3.3.1 Programming Model

BPEL4WS focus on combining existing web services into more sophisticated web services, where the composition of web services is specified in terms of the operations that need to be invoked, their ordering, and how to handle exceptional situations [74]. BPEL4WS, as other web services composition languages, is *process-based*, i.e. a process defines the order of interactions between the involved web services (control flow) and rules for data transfer between the invocations (data flow).

AO4BPEL proposes an extension of *process-oriented* composition languages with aspect-oriented modularity mechanisms. We extend BPEL with aspects, pointcuts, and advice. In the case of AO4BPEL, the base program is a BPEL process and the weaver is an aspect-aware orchestration engine.

Joinpoint Model and Configuration of Pointcuts

In AO4BPEL joinpoints are well-defined points in the execution of BPEL processes. Since web services are always described by their interfaces and are always accessed

¹ Web Page, <http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AO4BPEL/index.html>

only through the information provided in such interfaces, we can consider that the joinpoint model of AO4BPEL is non-invasive.

BPEL processes consist of a set of activities, and therefore, each activity in a BPEL process is a potential joinpoint in AO4BPEL. The current prototype supports the <invoke> and <reply> joinpoints. There are *primitive* and *structured* activities in BPEL. Structured activities act as containers for other activities according to some predefined control flow patterns such as sequential and parallel execution. Examples of primitive activities are: <invoke>, which specifies that a web service operation has to be invoked; <receive>, which specifies that the process must wait until the client calls an operation, and <reply>, which specifies that the process has to send a message. Examples of structured activities are: <sequence>, which specifies a list of sequential primitive activities, and <while> and <switch>, which behaviour is similar to while and switch in a programming language.

The pointcut language of AO4BPEL is XPath. Since BPEL processes are expressed as XML documents, the use of XPath allows easily selecting the activities where the behaviour of AO4BPEL aspects must be added. In an AO4BPEL aspect, the element <pointcut> is an XPath expression that selects the activities where aspects must be evaluated. In order to define these expressions it is possible to use the attributes of the BPEL activities as predicates to choose relevant joinpoints.

Aspect Advice in AO4BPEL

An aspect in AO4BPEL is described in an XML document and includes the definition of both aspect advice and pointcuts. An advice in AO4BPEL is a BPEL activity that specifies some crosscutting behaviour that should be executed at certain joinpoints. It can be a primitive or a structured activity. Like AspectJ [137], AO4BPEL supports before, after and around advice. However, since workflow languages support more patterns than in programming languages (e.g. split, join, parallelism, etc.), the around advice can be used to implement new kinds of advice e.g. the parallel advice.

Aspect Instantiation Mode

AO4BPEL is a dynamic AOP language where aspects are woven/un-woven with the processes at runtime. AO4BPEL supports process-level aspect deployment and instance-level aspect deployment. In process-level aspect deployment, the aspect applies to all instances of a process, whereas in instance-level aspect deployment it applies to specific instances based on, for example, the value of some variables.

3.3.2 Platform Infrastructure and Services

AO4BPEL supports dynamic weaving, i.e. aspects can be deployed or un-deployed at process interpretation time. The supporting platform infrastructure is based on an extension of the BPEL orchestration engine in order to produce and aspect-aware BPEL orchestration engine.

Infrastructure of AO4BPEL

The architecture of the aspect-aware BPEL orchestration engine of AO4BPEL is shown in figure 2.

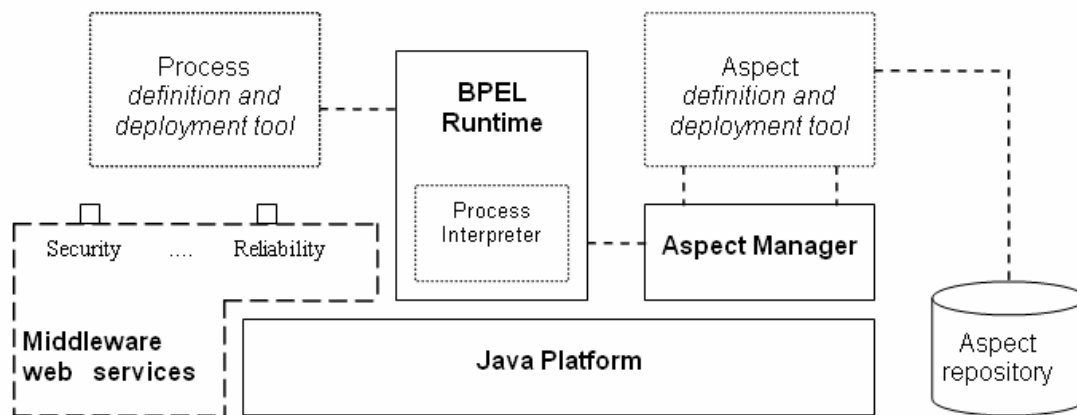


Figure 2. Architecture of the AO4BPEL Web Service Composition System (taken from [70])

There are five main components in this architecture:

1. The *process definition and deployment* component manages the registration of components.
2. The *BPEL runtime* is one of the core components and is an extended process interpreter that manages process instances and message routing. It also checks if there is an aspect before or after the interpretation of each activity. When an aspect must be executed the control is passed to the aspect manager that executes the advice and then returns the control to this component.
3. The *aspect definition and deployment* component manages the registration and activation of aspects. It is implemented as a web application using Java Server Pages.
4. The *aspect manager* controls the aspect execution.
5. The *middleware web services* are used to provide support for non-functional concerns in web service composition such as security, persistence, reliability, and transaction. For example, the security service can be used to make an invoke activity more secure by encrypting or signing the respective SOAP message.

Services of AO4BPEL

From the point of view of the software developer, the main services provided by the platform to final users are the registration and deployment of both processes and aspects into the AO4BPEL execution engine. Aspects can be deployed dynamically at runtime whilst BPEL processes are running. In addition, AO4BPEL aspects have been used to implement a light-weight process container for the integration of middleware services in BPEL web service composition as shown in figure 3.

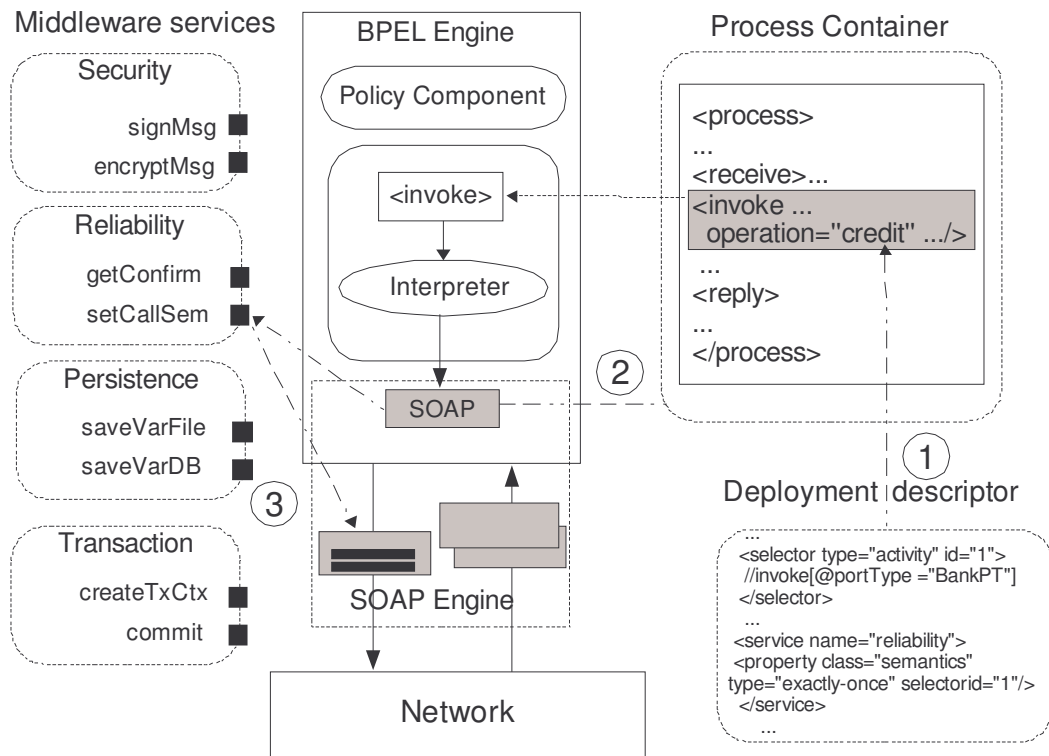


Figure 3. AO4BPEL middleware services

AO4BPEL security aspects can define a pointcut that selects all activities where processes interact with a certain partner, and an advice that adds authorisation functionality at these points by calling the security middleware web service. A framework has been developed for the integration of middleware concerns in web service compositions. The main components of this framework are the process container, middleware services and the deployment descriptor. The deployment descriptor selects BPEL activities with shared non-functional requirements. These activities will be intercepted by the container. The container is a light-weight process container implemented using a set of AO4BPEL aspects. It plugs-in calls to dedicated middleware web services. Services for security, reliability, persistence, and transactions have been developed for BPEL. The middleware services are based on web service specifications such as WS-Security and WS-Reliability whenever available and appropriate.

3.3.3 Current Status of Development

Currently, there is a prototype of AO4BPEL that supports the <invoke>, <reply>, and <receive> joinpoints. This prototype is based on IBM's BPWS4J orchestration engine. The current prototype supports before and after advice and the authors state that these are the first joinpoints to implement because basic activities represent the interaction points of the composition with the external partners.

One of the current uses of AO4BPEL has been the separation of business rules from the process specification, by implementing business rules based on aspects. Another usage is the integration of middleware concerns with BPEL to provide better support for reliability, security, transaction and persistence in BPEL compositions. Although

BPEL is becoming the standard language for web service composition, this approach is not specific to BPEL and may be applied to any process-oriented composition language supporting business processes.

Future plans include generalising the idea of AO4BPEL to allow for workflow modelling. In this manner aspects can be used to capture crosscutting concerns in a modular and separate way in workflow specifications.

3.3.4 Evaluation

AO4BPEL is one of the few applications of the AOP paradigm outside the context of programming languages. In this approach, the *flexibility* and adaptability of web service composition is improved by using an aspect-oriented approach. Aspects modularise crosscutting concerns and they can also be used to change the behaviour of the composition dynamically. Using aspects together with BPEL processes makes change and especially crosscutting change more easy as it provides pointcuts for better quantification. Since AO4BPEL aspects can be plugged in and out, certain features can be flexibly switched on and off within the composition.

Regarding *performance*, the most important activities in BPEL are the messaging activities invoke, receive, and reply. Each of them corresponds to one or more SOAP messages that are sent over the Internet. For this reason, the cost of checking whether an aspect applies to the current activity or not is negligible compared with the cost of an interaction with a remote web service. From this perspective, AO4BPEL has a quite different usage context than aspect-oriented programming languages.

With regard to *reliability*, the weaving mechanism of AO4BPEL modifies the activity lifecycle implemented by the BPEL orchestration engine. The interpretation flow is modified whilst taking into account all BPEL specifics such as links and fault handling. In addition, AO4BOPEL aspects have been successfully used to improve and monitor quality of service in web service compositions. Aspects can be written to measure the response times of a partner service and also integrate the BPEL process with reliability middleware for web services according to WS-Reliability. The reliability aspects make BPEL messaging activities more reliable by eliminating message loss and message duplication, thus leading to web service interactions with the exactly once semantics

There are some approaches for dynamic web service composition that use the BPEL extensibility mechanism to allow for adaptability. However, such approaches do not support quantification, in that the adaptability constructs must be specified in a point-wise in all processes and activities where they are needed. These constructs are also tightly-coupled with the process specification and cannot be switched on and off in a flexible manner.

Since workflow specifications always must be interpreted by a workflow engine, it is easier to implement a weaver for an aspect-oriented workflow language. One has just to modify the activity lifecycle implemented by the engine by checking if any aspects apply before and after the activity at hand.

Finally, regarding the criteria used to compare middleware platforms for AOSD from the point of view of the software developer, AO4BPEL is a non invasive approach specifically developed for the separation of aspects in the composition of Web Services. In AO4BEL aspects are described in XML and include both advice and pointcuts, reducing the reusability of aspect advice in different contexts. AO4BEL supports adaptability of final applications making possible to (un)deploy aspects at runtime.

3.4 AspectJ2EE

AspectJ2EE [75] is an aspect language that can be used to apply aspect-oriented programming to the development of enterprise applications. This approach is being developed by the Department of Computer Science of the Israel Institute of Technology (Technion).

3.4.1 Programming Model

AspectJ2EE does not define a new programming model. Instead, AspectJ2EE maintains the standard object model and, therefore, aspects are applied to objects. In AspectJ2EE, aspects are not applied globally but only to explicitly selected classes. These classes are considered to be the application's Enterprise Beans, in J2EE terminology.

Joinpoint Model and Configuration of Pointcuts.

AspectJ2EE supports a rich set of joinpoints, including method and constructor execution, field read and write access, object and class initialisation, remote call, control-flow based. Concretely, "remote call" is a new kind of joinpoint introduced by AspectJ2EE that applies to remote calls to methods while local calls are unaffected. This joinpoint is implemented in AspectJ2EE by affecting the stub generated at deploy time for use by EJB clients.

The joinpoint model in AspectJ2EE is a non-intrusive model, in the sense that existing classes are never changed. Changes are made, by means of sub-classing, only to the target classes, and never to their clients. In other words, those classes that use the services offered by the enterprise beans are not modified. It is an invasive-model in the sense that changes to the enterprise beans can include changes to any method (including private methods) as well as internal access to any field (including private fields). By "internal access" the authors mean that access to a non-private field by some other class would not be trapped.

In AspectJ2EE, the application of aspects is defined *per bean class*. This means that during the deployment of the application, the application assembler has to define the number of aspects that will be applied to each bean, and the order of application using a *deployment descriptor*. They extend the traditional deployment descriptor syntax (used to specify the application of services to EJBs in J2EE) with new elements: (1) the <aspect> element to define each aspect applied to a bean; (2) the <pointcut> element to bind any abstract pointcut, and (3) the <value> element for specifying the initial values of fields. The <pointcut> and the <value> elements are used to provide

parameters to abstract aspects, which can define abstract pointcuts and abstract fields (a field whose initial value is specified at application time).

The order of precedence coincides with the order of the specification of aspects in the deployment descriptor. Since aspects have to be defined for each bean, AspectJ2EE allows a different order or precedence for the same set of aspects when they are applied to different beans.

In conclusion, although in AspectJ2EE the configuration of pointcuts can be directly hard coded in the aspect class, it is also possible to define abstract aspects, where pointcuts are described in the deployment descriptors. Abstract aspects will then be parameterised at application time with this information, making aspects more reusable in different contexts.

Aspect Advice in AspectJ2EE

An aspect in AspectJ2EE is similar to a generic class. For instance, class *Persistence<Bean>* is the conceptual equivalent of applying aspect *Persistence* to the EBJ named *Bean*. The difference with a generic class is that the body of an AspectJ2EE aspect includes the definition of pointcuts and advice. It defines “before”, “after”, “after returning”, “after throwing” and “around” advice. Therefore, the body of an aspect in AspectJ2EE looks very similar to the body of an aspect in AspectJ.

Aspect Instantiation Mode

As mentioned before, in AspectJ2EE aspects are defined per bean class. Conceptually, a new advised class is generated during the deployment of the application for each aspect that has to be applied to a bean. This class will inherit a previously generated advised class or the main bean class, depending on the order of application of aspects. Technically, all this sub-classing can be collapsed into a single, rich subclass generated per bean class.

Regarding the distribution of aspects, they will normally live on the server side of the application server, similarly to the fixed set of services traditionally offered by application servers. However, using the *remotecall* joinpoint, AspectJ2EE also allows the separation of some kinds of aspects that crosscut both the client and the server side of an application. By adding code both at the sending (in the stub) and receiving ends of remotely-invoked methods, they are able to create an additional layer in the communication stack. For instance, this will allow the separation of aspects such as encryption or compression. The encryption code would be added at the stub and the decryption code at the remote tie. The same can be done to compress information at the client site and decompress it at the server site.

3.4.2 Platform Infrastructure and Services

AspectJ2EE relies on the J2EE application server architecture. The AspectJ2EE weaver weaves the base class together with its aspects using the same mechanisms that J2EE application servers apply to combine services with the business logic of enterprise beans. It uses standard Java language and an unmodified JVM.

Infrastructure of AspectJ2EE

The only difference between J2EE and AspectJ2EE regarding the application server infrastructure is that AspectJ2EE introduces a *deploy-time* weaver, which weaves aspects and the core EJB components during the deployment of the application. The idea behind the deployment-time weaving is to extend the mechanism used by J2EE application servers to incorporate the use of J2EE services into components during the deployment phase, to be able to apply any kind of aspects, not only those services provided by the platform vendor.

Using this weaver, AspectJ2EE generates new classes that inherit from the core program classes. This means that AspectJ2EE uses a non-intrusive weaving mechanism, where the source and the binary code of the core components are not modified, and can be executed on any standard JVM.

For each application of an aspect to a class the deployment tool generates an *advised class*. The generation of this class is governed by the advice that were provided in the aspect. All the fields, methods and inner classes that were defined in the aspect are copied to its advised class. The information in the advised class is generated automatically based on the advices in the aspect.

When more than one aspect has to be applied to a bean, the sequence of aspects to be applied is conceptually equivalent to a chain of inheritance, with the chain starting with the main bean class.

Services of AspectJ2EE

Using AspectJ2EE, the fixed set of standard J2EE services (e.g. persistence, transaction management, security, load balancing, etc.) is replaced by a library of core aspects. These services can be extended with new ones such as logging and performance monitoring, encryption, data compression and memoisation. Some of these aspects can even crosscut several tiers in multi-tier applications.

3.4.3 Current State of Development

Work is currently under development to implement AspectJ2EE as a new deployment process for the IBM WebSphere Application Server, version 5.0. They plan to use AspectJ2EE to define every service currently provided by WebSphere as an aspect.

3.4.4 Evaluation

AspectJ2EE relies on the J2EE application server as middleware infrastructure and therefore the evaluation in section 2.1.1. is completely applicable to AspectJ2EE. The only interesting difference is that using AspectJ2EE it is possible to cope with the lack of flexibility resulting from the fixed set of services offered by the J2EE container.

In addition, the weaving of aspects and EJB components is performed at deployment time and therefore, although there are no performance studies reported by authors, the performance of J2EE applications should not be affected by the use of AspectJ2EE aspects.

Regarding the criteria used to compare middleware platforms for AOSD from the point of view of the software developer, AspectJ2EE is a non-intrusive and invasive approach. AspectJ2EE was specifically developed for the separation of aspects in J2EE applications, and where aspects are weaved with EJB components at deployment time. AspectJ2EE is geared towards reusability of aspects. Aspects with abstract pointcut definition can be applied to different classes, without having to explicitly subclass each aspect (as in AspectJ and similar languages); the concrete specification of the pointcut is defined as an aspect and is applied to each specific class, in the deployment descriptor. Finally, in AspectJ2EE not only the definition of pointcuts but also the binding between pointcuts and advice can be defined in separate XML deployment descriptors which increases the adaptability of final applications during the first phases of development and before the application is deployed in the application server. Once the application is deployed, the binding between aspects and components can not be adapted neither at load time nor at runtime. However, it is possible to define multiple advised versions of the same bean (e.g. Secure<Account>, Logging<Account>, and Secure<Logging<Account>>), and to choose which to use at runtime.

3.5 AspectWerkz²

AspectWerkz 2 [76][77][78][79] is a dynamic, lightweight AOP framework for Java. It is a Java extension written in pure Java to support AOP. AspectWerkz is open source software sponsored by BEA Systems. In the rest of the section we refer to the features in version 2 of AspectWerkz.

3.5.1 Programming Model

AspectWerkz does not define a new programming model. Aspects, advice and introductions in AspectWerkz are written in plain Java and the target classes can also be regular Plain Old Java Objects (POJOs).

Although AspectWerkz by itself cannot be considered a middleware platform for AOSD, it enables the integration of AO applications with enterprise solutions using application servers and J2EE facilities.

Join Point Model and Configuration of Pointcuts

AspectWerkz defines a rich and highly orthogonal join point model. Multiple interception points are defined, allowing the catching of method and constructor executions and calls, field getting and setting, control flow and exception execution. An invasive model is used, being able to catch many points inside an object.

² Web Page, <http://aspectwerkz.codehaus.org/>

In AspectWerkz, pointcuts are defined using declarative expressions. The *execution*, *get*, *set*, *cflow*, *call* and *handler* expressions are used to identify method and constructor execution, field access and customisation, method (or constructor) calling from a class and exception execution. Wildcards are supported: * matching exactly one type or package, and ‘..’ matching 0 or more types or packages. + is used to indicate subtypes, so ‘myClass+’ means ‘the class (or interface) myClass or any class that extend (or implements) myClass’. Declarations of pointcuts are made using patterns. Complex pointcuts can be constructed using composition using logical operators (and ‘&&’, or ‘||’ and not ‘!’).

Aspectwerkz provides two different ways of declaring pointcuts: inside an aspect declaration by means of annotations (self-defined aspects) or in a separate XML file (XML-defined aspects) using a XML deployment descriptor. Self-defined aspects contain advice and pointcut declaration together. Pointcuts are named, and later bound to advice using that name. However, aspect reuse can be achieved using an extension mechanism of Java classes (i.e. aspects are pure Java classes). Common aspect behaviour is encapsulated in an abstract class, without defining any concrete pointcut, just the names. In another class extending the abstract class, we define only our concrete pointcuts, and use the advice defined by the abstract class.

XML-defined aspects make aspects easily reusable, separating aspects and pointcuts, declaring pointcuts and binding aspects to advice in a separate XML file, a deployment description file. This type gets a more loosely coupled model, increasing reusability.

Aspect Advice in AspectWerkz

An aspect is implemented like a regular Java class, which does not have to extend any special class or implement a specific interface. Instead, it can extend any class it needs. The only requirement is that the aspect needs to have either no constructor (just the default one) or one of these two different constructors defined:

- A default no-argument constructor - only needed if the aspect has other constructors that take parameters, apart from the one defined below.
- A constructor that takes an *AspectContext* instance as its only parameter, used to access contextual information.

AspectWerkz implements around, before and after, after finally, after returning and after throwing (to catch exceptions throwing) advice. They are implemented as regular methods in Java. Each of these methods takes a unique parameter: the join point instance that has thrown the aspect execution. A *proceed* method is available on the join point instance. A call to this method invokes the next advice in the chain or continues with normal execution at join point. Information about join points can be retrieved using the RTTI (*Run -Time Type Information*) property and using the method of Join Point class *getSignature()*, whihc provides the signature of the current join point.

In addition to aspects, AspectWerkz defines *introductions*, which allows the addition of code to existing classes. They are implemented using *mixins*, which are inner

classes of an aspect and consist of one or more interfaces and one or more implementations of those interfaces.

An aspect can be defined by either XML definition or annotations. They are two different views of the same underlying model. Using annotations (Java 5 annotations or JavaDoc annotations) aspects can be defined as metadata attached to regular Java elements. This is done by declaring information about aspect deployment, introductions, binding of advice and pointcuts, advice behaviour and mixins. On the other hand, all these metadata can be placed in a external XML file, where pointcuts are defined (they do not appear in the Java class implementing the aspect), pointcuts and advice are bound, introductions and mixins are declared and deployment information is provided.

Aspect Instantiation Mode

There are four models of deployment for aspects: *perJVM* (one instance of an aspect by JVM, a singleton implementation), *perClass* (one instance per target class), *perInstance* (one instance per instance of target class). Information about deployment can be written either in the aspect unit (like metadata using annotations) or in the XML deployment descriptor. An XML deployment descriptor is needed to activate the aspects, whether pointcuts are defined inside aspects or not.

3.5.2 Platform Infrastructure and Services

AspectWerkz offers a rich set of weaving options. Aspects can be woven with plain objects by using byte-code modification at compile-time, known as *offline weaving*, and at load-time and runtime, both known as *online weaving*. Runtime weaving is the most powerful and customisable weaving mechanism offered by AspectWerkz. Using this mechanism software developers have the possibility to add, remove and re-structure advice as well as swapping the implementation of introductions at runtime. Only the addition of new pointcuts is not currently supported by AspectWerkz.

Infrastructure of AspectWerkz

In this document, which is focused on middleware platforms, we are interested in the integration of AspectWerkz with enterprise application servers. In this context, AspectWerkz can be considered as a J2EE extension that is used over plain Java objects working at class level. Currently, AspectWerkz has been integrated with WebLogic [134], JBoss (see section 3.8), Tomcat [135] and WepSphere [136].

If the weaving is static then no middleware infrastructure is needed and, therefore, no more than a common JVM is required. Otherwise, if the weaving is online, the AspectWerkz weaving mechanism lives in the system class loader as a hook. This architecture is specific to the use of HotSwap to perform load time weaving. Other possibilities offered by AspectWerkz are hooking in at the level of the Java class loader, with the module BEA JRockit or using a command line tool.

In addition, AspectWerkz includes the concept of *Aspect Container* that can be integrated as part of the application server as shown in figure 4. The aspect container manages AspectWerkz aspects, being responsible for creating and configuring them.

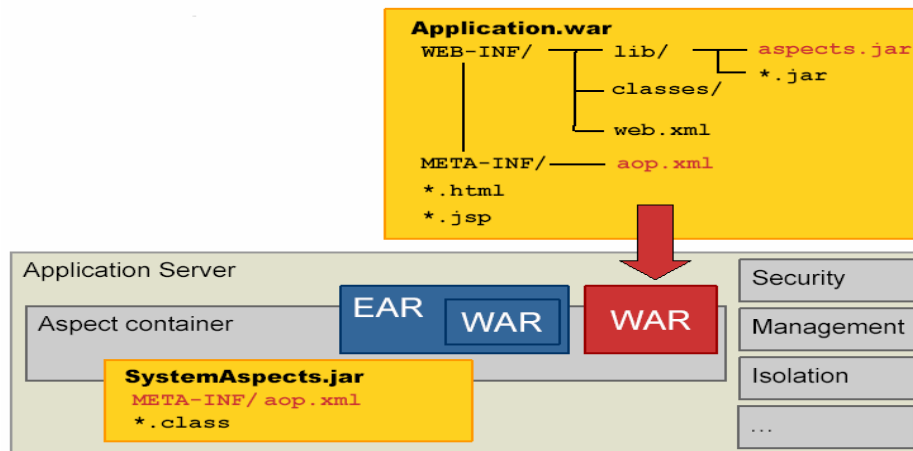


Figure 4. Integration of AspectWerkz in an Application Server (taken from <http://aspectwerkz.codehaus.org>)

In version 2, AspectWerkz introduces an *Extensible Aspect Container* that can weave, deploy and run any aspect (e.g. AspectJ, Spring (see section 3.12), AOP Alliance [138] and AspectWerkz aspects) no matter how it is implemented and defined. Basically it allows the plugging in of extensions that can handle the specific framework details, while the common things are shared. The container is responsible for handling the semantic differences between the different aspect models to allow the aspects to coexist in one single runtime environment.

Services of the AspectWerkz Aspect Container

The main service offered by the aspect container is to package aspects inside a WAR or an EAR file and to deploy them in a similar way to that of the rest of J2EE software artefacts, e.g. servlets. Other services offered by the aspect container are the management of aspects, security, isolation, hot deployment and undeployment, etc as shown in figure 4.

3.5.3 Current Status of Development

Currently version 2 of AspectWerkz has been released. Using AspectWerkz aspect-oriented applications can be implemented in Java version 1.3/1.4 and the latest version 1.5. Therefore, Aspectwerkz uses the most recent features in Java (e.g. annotations), while it keeps compatibility with old JVM's providing the mechanism to use the new features in some way in old JVM's.

Regarding the weaving mechanism, AspectWerkz provides static and dynamic weaving. The dynamic weaving process can be performed following four alternative strategies: (1) using JSR-163 JVMTI facility of Java5, (2) using Hotswap (you can choose among three different ways of using Hotswap), (3) modifying the

bootclasspath for old JVM not supporting Hotswap, and (4) using a specific extension of JRockit JVM [134] based on *Java Management API (JMAPI)*. The JRockit extension allows running AspectWerkz without changes at the ClassLoader level. AspectWerkz can be integrated with several J2EE application servers, such as BEA Weblogic, JBoss, WebSphere, etc.

Finally, several supporting tools are also available such as a plug-in for the Maven [139] build system and an IDE for deploying aspects. Information about how to integrate AspectWerkz in a different IDE is provided, including integration in NetBeans 3.6 [140] and integration in Eclipse [141] with and without the AspectWerkz Eclipse plug-in.

3.5.4 Evaluation

Here we are interested in the evaluation of AspectWerkz 2 when it is integrated within a J2EE application server. With respect to the evaluation of the middleware infrastructure, the discussion in section 2.1.1 is applicable here with some differences. One of them is the performance of the final application. Since Aspectwerkz needs to weave aspects at runtime with core Java classes, the weaving mechanism may bring some overload during the execution time. In order to solve this possible limitation, AspectWerkz offers a really efficient weaving mechanism, minimizing this overload.

In this sense, the AspectWerkz weaver in version 2 is completely based on static compilation without using reflection, making the code faster. Authors have released a micro benchmark³ using the AspectWerkz version 2 and the extensible aspect container. Among other things they have compared the execution of different kinds of advices using AspectWerkz, and other application servers such as JBoss 1.0, with the execution of non-advised code. The conclusion of this study is that the overload introduced by AspectWerkz is between 5 and 75 nanoseconds per interaction depending on the kind of advice, while in JBoss it is much higher (130-290).

AspectWerkz is extensible with respect to the aspect container. In version 2 they have introduced a very open and extensible container that allows plugging in different *Aspect Model Extensions* to run aspects from other AOP frameworks inside the AspectWerkz container. Currently they offer an extension that supports aspects that implements the *AOP Alliance* interfaces including Spring, JAC (see section 3.7), etc. In addition, aspect-oriented software developers can implement their own plug-ins to add new aspect models to the container. Furthermore, they even have the possibility of providing their own aspect container implementation, in case they need to control how their aspects are instantiated. Also the online weaving mechanism can be modified, it being possible to provide other developers' own byte-code transformation at load time.

Regarding *flexibility*, the separation of aspects using AspectWerkz increases the number of common services that can be used by final applications. However, the intention of AspectWerkz, at least up to now, has not been to define a homogeneous model that allows separating using the same mechanism both the common services

³ Web page http://aspectwerkz.codehaus.org/new_features_in_2_0.html

usually provided by the application server container and the new services separated with AOP.

Finally, according to the criteria used to compare middleware platforms for AOSD, AspectWerkz is an invasive approach developed to apply aspects to object-oriented applications. Although it can be integrated with a J2EE application server no specific support to apply aspects to EJB components is provided. AspectWerkz is very flexible in the sense that it offers different kinds of weaving mechanisms at compilation, loading and even at runtime. Aspects can be developed to be reused in different contexts, though there is also another possibility where aspect's pointcuts are directly hard coded together with the aspect advice using annotations, resulting in more coupling aspects. Therefore, the reusability of aspects depends on the software developer. Finally, in XML-based aspects not only the definition of pointcuts but also the binding between pointcuts and advice can be defined in separate XML files which increases the adaptability of final applications during the design. In addition, the possibility of hot-deploys and undeploys aspects at runtime increases the extensibility and adaptability of applications also at runtime.

3.6 CAM/DAOP⁴

CAM/DAOP [80][81][82][83] is a component and aspect based approach that combines the benefits of both CBSD and AOSD disciplines. CAM (Component-Aspect Model) is a new component and aspect model. The underlying platform supporting the CAM model is the DAOP (Dynamic Aspect Oriented Platform) platform. CAM/DAOP has been developed by the Languages and Computer Science Department at the University of Málaga.

3.6.1 Programming Model

The CAM model is a new component and aspect model that has been defined from scratch, though taking into account some well considered features of standard component models such as EJB and Corba Component Model (CCM). The main entities of the CAM model are components and aspects, which are defined as first-order entities, together with a non-intrusive composition mechanism for plugging aspects into components.

In CAM, aspects are treated as a “special” kind of component and, consequently, both share a common definition. Following the definition of component by Szyperski, CAM considers both components and aspects as coarse-grained encapsulated entities that act as units of composition with contractually specified interfaces and explicit context dependencies. They may be deployed independently and are subject to third-party composition.

Similarly to other component models such as CCM, CAM describes the interfaces of components and aspects using an interface description language (IDL). A component's IDL describes not only the services the component provides to the environment, as is usual in component platforms (the *provided* interface), but also

⁴ Web Page <http://www.lcc.uma.es/~pinto/cam-daop.html>

those services it requires in its interaction with other components (the *required* interface). How to evaluate aspects is described in an *evaluated* interface, instead of a provided interface as for components, which includes the joinpoints that an aspect is able to intercept and evaluate. In CAM, *aspect evaluation* means the execution of the corresponding aspect advice. In CAM these IDLs are part of an XML-based architectural description language (DAOP-ADL) used to describe components and aspects, together with the composition rules that govern the weaving of components and aspects.

The other important feature of the CAM model is the way in which the entities of the model communicate among themselves. Following the standard practices of CBSD, components interact by exchanging *messages* and by throwing *events*. CAM understands the meaning of messages and events in the CBSD sense: messages are sent to a specific target entity and events are messages with no information about the target component. The handling of events is resolved at runtime by a *coordination* aspect that encapsulates a component interaction protocol that will decide which are the target components of a given event, at runtime.

A relevant feature of the CAM model that we introduce here, since it is needed to explain the rest of features of the CAM/DAOP approach, is the concept of *role name*. CAM assigns a unique role name to identify both components and aspects. A role name identifies a specific functional or extra-functional property and will be played by a component or an aspect that implements this property. These role names are architectural names that will be used for component-aspect composition and interaction, allowing loosely coupled communication among them -- i.e. no hard-coded references need to be used for exchanging information, but just a role name identifying the target of a message. In addition, if there are several instances of a component playing the same role inside a distributed application they can be distinguished by the *role instance name*.

Joinpoint Model and Configuration of Pointcuts

CAM considers that aspects are applied to black-box components. Therefore, CAM intentionally avoids the definition of joinpoints that intercept the internal behaviour of a component, as we only have access to a component through its public interface. Thus, CAM defines a non-invasive joinpoint model where aspects can be applied before and after (incoming and outgoing) messages and events, and also before and after the creation and destruction of component instances.

In CAM/DAOP, aspect pointcuts are described in an XML-based document outside the component and aspect definition. They define when and how to apply aspects to components and are expressed in terms of the following elements: the role names of the source and the target components affected by the rule, the signature of the messages from which the joinpoint was reached, the signature of the messages the rule is applied to, and a list of aspect role names that specifies the role names of the aspects that will be evaluated. This list is implemented using a bi-dimensional array of strings with the format $\{\{A1\},\{A2\},\{A3,A4\}\}$ where every A_i is an aspect role name. This bi-dimensional structure allows CAM/DAOP developers to specify two kinds of aspect evaluation: sequential evaluation and parallel evaluation. Aspects enclosed in the outer brackets, for instance $A1$ and $A2$, are evaluated sequentially. On the other

hand, aspects in the inner brackets, for instance A3 and A4, will be evaluated concurrently.

Finally, an aspect evaluation rule specifies the moment in which aspects are applied (the *when* attribute). The available joinpoints are BEFORE_SEND, AFTER_SEND, BEFORE_RECEIVE and AFTER_RECEIVE to apply aspects before and after sending and receiving a message or an event, and BEFORE_NEW, AFTER_NEW, BEFORE_DESTROY and AFTER_DESTROY, to apply aspects before and after creating or destroying a component instance. Although CAM defines “before” and “after” rules, CAM aspects can also be applied “around” a joinpoint as other approaches do. Since CAM coordination aspects can send messages to components, they can replace or modify component interactions forwarding messages to a different target and even running additional code.

Aspect Advice in CAM/DAOP

An aspect in CAM/DAOP defines the aspect advice. CAM/DAOP provides one different method for each of the joinpoints that can be intercepted (evalBEFORE_SEND(), evalBEFORE_RECEIVE(), evalSEND_EVENT(), etc...). In addition it offers a general *eval()* method that is independent from any joinpoint. Encapsulating the aspect advice in this special method, the aspect advice is general enough to be executed at any joinpoint that can be captured in the system.

CAM aspects have access to the information about the intercepted joinpoint. This includes information about the source and target components, the message from which the execution of the intercepted joinpoint was initiated, the intercepted message, event, component initialisation or component finalisation joinpoint and the parameters of the message/event. It has also a reference to the DAOP platform in order to have access to the services offered by the platform.

Component and Aspect Instantiation Mode

CAM components can be *local* or *remote*. Local components are created in the local instance of the DAOP platform. Remote components are instantiated in a specific DAOP platform's URL. This information is described as part of the deployment information of CAM components.

Regarding the number of aspect instances, the aspect deployment information in CAM/DAOP describes whether the DAOP platform creates: (1) only one instance of an aspect shared by all the DAOP platform instances connected to the same application (an *environment-oriented* aspect), providing a centralised aspect; (2) one instance of the aspect for each instance of a DAOP platform (a *user-oriented* aspect), providing a distributed aspect; (3) an aspect instance that is shared by all components playing the same role (a *role-oriented* instance), and (4) an aspect instance for all the components sharing the same role and the same role instance name (a *roleinstance-oriented* aspect).

Other deployment information for aspect is *criticality*. CAM aspects are classified as *critical* aspects and *non-critical* aspects, depending on how important the result of the evaluation of that aspect is to continue or not the application execution. For *critical*

In this section we explain the internal structure of the DAOP platform (the elements that appear below the DAOP Platform class in figure 5). Basically, the DAOP platform arranges its internal information in two objects. The information contained in these objects is used to implement the services offered by the DAOP platform.

1. The *ApplicationArchitecture* class and its subclasses store the architectural description of the application. This information is specified in an XML-based document using the DAOP-ADL language. Using this language, the CAM model of the application is transformed to a set of XML documents that can be easily interpreted by the DAOP platform at runtime to perform the dynamic weaving of components and aspects.
2. The *ApplicationContext* object holds the references of all the components and aspects currently instantiated for a specific application. Thus, the *ApplicationContext* class is like a name service that links component and aspect instances with their role names and role instance names.

Since for each user who is connected to a CAM/DAOP application an instance of the DAOP platform is created, these instances must implement the *RemoteDAOPPlatform* interface. This interface defines the methods needed to allow the communication among the different instances of the DAOP platform.

Services of the DAOP Platform

Similar to other component platforms, the DAOP platform provides a set of common services to develop distributed applications (the elements that appear above the DAOP Platform class in figure 5). The DAOP platform uses the information stored in the *ApplicationArchitecture* object and the *ApplicationContext* described above to provide all these services.

1. *Component and Aspect Instantiation*. DAOP components and aspects can create or destroy other components using the corresponding methods of the *ComponentFactory* interface (see figure 5Figure 5). Components are identified by a role and a role instance name. Thus, a software component just needs to specify strings with the role name and the instance name, and never its implementation class. Likewise, aspects are also created or destroyed, but only by the platform using a similar interface that also identifies aspects by their role name, which in this case is private and only used by the DAOP platform.
2. *Component Communication and Coordination Service*. As in other component platforms, DAOP allows components to send synchronous and asynchronous messages, as well as to broadcast a message to several targets. DAOP also allows components to throw events to other components. DAOP implements four primitives in order to send messages and events between components (see the *CommunicationService* interface and all its subinterfaces in figure 5). Communication by events is very useful to decouple components, and is specially suited to enable (re)use. By intercepting the throwing of events, the DAOP platform provides a joinpoint that occurs within the execution of a component method. The location (local/remote) of the target component or a message or an

event is transparent to the source component and is resolved at runtime by the DAOP platform.

3. *Aspect Evaluation.* The component composition mechanism described above is extended in DAOP to incorporate dynamic evaluation of aspects. When a component creates or finalises other components or sends a message or an event using any DAOP communication primitives, the DAOP platform intercepts it and evaluates the corresponding aspects. DAOP aspects should implement the *AspectEvaluationService* interface that includes the different advice methods.
4. *Property Storage.* The CAM model defines the concept of property to solve data dependencies between entities of the CAM model. Using this service a producer entity will set the value of a property with the *setProperty(String propertyname, Object value)* method of the *PropertyService* interface in figure 5, and later a consumer entity of that property will get its current value with the *getProperty(String propertyname)* method. Property instances are stored in the *ApplicationContext* object of the DAOP platform.
5. *Persistence.* The *PersistenceService* interface provides the functionality required to store and retrieve the current state of components. This service may be used to implement a persistence aspect that simply has to invoke the `storeComponent(CID component)` and `retrieveComponent(CID component)` methods. The implementation of these methods serialises or deserialises the attributes that are part of the state of a component (see the `State` class of the CAM model of Figure~\ref{fig:CAMmodel}), and then store or retrieve them in a data repository of the DAOP platform.
6. *Application Architecture Configuration.* The *AAConfigurationService* interface provides a set of methods to modify at runtime the software architecture of the application, which is stored in the *ApplicationArchitecture* object as described before. It is possible to add, modify or remove the description of components, aspects, properties, or even composition rules using the corresponding methods of the *AAConfigurationService* interface.

3.6.3 Current Status of Development

CAM/DAOP was designed to be independent from any supporting language and distributed object platform. Currently, CAM/DAOP has been implemented based on Java/RMI as the base communication mechanism and the Java reflective package for dynamic composition. Users initiate CAM/DAOP applications downloading an application applet through a DAOP Application Directory. An instance of the distributed DAOP platform is created at each user site during applet downloading.

In the current implementation of CAM/DAOP components and aspects are plain Java code. Software developers do not have to manage any matter related to the implementation of remote objects, such as the definition of remote interfaces and implementations, generation and distribution of stubs and skeletons, etc.

The CAM/DAOP prototype also supports the use of the DAOP-ADL language to describe the architecture of applications. The process of describing and validating the

application architecture is semi-automatic, as they provide a set of tools that support the software architect's task. In addition, in order to integrate the DAOP-ADL language into the platform they are currently developing tools that automatically generate DAOP-ADL descriptions of components and aspects by binary code inspection. Using this tool, CAM/DAOP simplifies the software developer's task by making it easier to plug components and aspects into an application.

New implementations of CAM/DAOP for CORBA and .NET are under development. Other ongoing work is the development of a static composition mechanism using the Java BCEL API. Extending the DAOP-ADL language to specify whether an aspect must be woven into components statically or dynamically, this tool will manipulate component class files to weave static aspects at compilation time. Following this approach, aspects that are composed dynamically at runtime will not invade the component code, while aspects that are composed statically will be part of the component code.

3.6.4 Evaluation

On the one hand, on the DAOP middleware platform side, CAM/DAOP does not provide any mechanism to extend the main services (communication, dynamic weaving, etc.) offered by the platform. Regarding the binding types it is flexible in the sense that it offers different kinds of communication mechanisms (asynchronous, synchronous and broadcast messaging and event service). Additionally, these communication mechanisms are all used in a homogenous way, which is an important advantage from the point of view of the usability of the platform by application developers. On the other hand, the CAM/DAOP kernel is extensible, being able to add new remote services that may be needed by applications. In the current implementation of CAM/DAOP, only the implementation of a RMI object and its register in the kernel using a kernel deployment file is needed.

Similar to the rest of middleware platforms for AOSD, CAM/DAOP increases the *flexibility* of the platform regarding the number and kind of crosscutting concerns that can be separated and used by application developers as it provides a homogenous model to separate any kind of crosscutting concern.

With respect to *performance* in CAM/DAOP neither components nor aspects need to be instrumented when they are loaded into the platform and therefore the performance of the final application can be affected only during the weaving of aspects at runtime. The authors have taken some measures using the Java/RMI implementation of CAM/DAOP to study the overhead that may be introduced by the runtime weaving mechanism and found the time to incorporate the evaluation of aspects at runtime is insignificant (around 20 ms). Since currently the applications developed on top of CAM/DAOP have been web-based applications, comparing this evaluation time with the time spent loading a web page from the same host, they are insignificant.

With respect to *reliability* CAM/DAOP assures the consistency of the composition between components and aspects by means of the DAOP-ADL language. Using this language, the weaving information is provided and validated during the description of the application architecture. Furthermore, this validated information is used directly by the platform to perform the dynamic composition of components and aspects. If

changes are performed at runtime, they are also validated following the same procedure. Additionally, CAM/DAOP establishes the order of evaluation of aspects, assuring that non-orthogonal aspects are not going to be executed concurrently. It also offers a property-based mechanism to resolve aspect dependencies.

Finally, according to the criteria used to compare middleware platforms for AOSD, CAM/DAOP is a non invasive approach developed to apply aspects to component-based applications. CAM/DAOP is very flexible regarding the variety of aspect instantiation modes and the different kinds of information about the deployment of aspects (e.g. criticality) that can be provided as part of the deployment information. CAM/DAOP promotes aspect reuse, avoiding explicit pointcuts and binding information in the aspect definition. Thus, aspects are independent of the components they affect, and can then be applied to different components at different times. These modifications can even be carried out at runtime, increasing the adaptability of final applications both at design and runtime.

3.7 JAC ⁵

JAC (Java Aspect Components) [84][85][86] is an aspect-oriented middleware written in Java. Its primary aim is to provide a set of concepts to enable distributed and dynamic AOP programming. Three levels of relationships exist between JAC and middleware: at the container level, at the aspect level, and at the programming model level. JAC is open-source software developed by the AOPSYS (TM) Company with the collaboration of the LIP6, the CEDRIC, and the LIFL/INRIA laboratories.

3.7.1 Programming Model

JAC is based on the use of containers, similarly to the J2EE middleware platform, although JAC implements its own standards. At the container level, JAC can be seen as a remotely accessible container for POJO and aspects. The architecture of this container includes a class loader, which is able to load application classes and aspects. Each host of a network involved in a distributed application with JAC must run the JAC container in daemon mode. This application server like architecture can be compared to J2EE, in that instead of having a set of hard-coded technical services, JAC technical services are programmed as aspects, and are only loaded if the application needs them. With this approach, the main goal of JAC is to solve the limitation of J2EE of providing just a fixed set of common services. With this approach, the main goal of JAC is to solve the limitations of J2EE in providing just a fixed set of common services.

The primary entities of JAC are *aspect components*, *wrappers* and *JAC containers*. The main responsibility of aspect components is to define pointcuts. Then, in wrappers the behaviour of aspects is encapsulated. Aspect components represent crosscutting properties which crosscut a set of base-objects and are hosted by JAC containers.

At the programming level model, JAC allows for the definition of distributed pointcuts. When defining a pointcut, the programmer can specify, in addition to the

⁵ Web Page, <http://jac.objectweb.org/>

expressions for selecting classes, methods and objects and a regular expression based on host names for selecting the hosts where the pointcut applies. Hence, the programmer can define pointcuts which includes joinpoints located on different remote hosts. This feature is handled by an aspect manager which is a service provided by the container. The aspect manager is an object that is replicated (with the replication aspect) on remote hosts. The different replicas are synchronised with the strong-consistency aspect. The definition of a pointcut with one of the aspect manager is propagated on all the replicas. Each replica can then decide whether the pointcut applies for its host. In JAC this is the base to develop distributed applications based on distributed aspects.

JAC provides a framework for AOP in Java. All concepts introduced by JAC are expressed in Java, without extending the language with new constructs. Using this framework a JAC application is composed of some classes implementing the core functionality and a set of aspect components. Classes implementing core functionality are POJO's, so you don't need any extra knowledge of Java platform to code them. Extra-functionality is added later by aspect components.

Joinpoint Model and Configuration of Pointcuts

The joint point model in JAC allows influencing the base program in three different ways:

- By extending base classes' semantics through metadata information that is attached to JAC Run-Time Type Information (RTTI). It can be tagged by any aspect component with some meta-data in order to extend its semantics.
- By implementing an internally defined MOP interface that allows the aspect components to react on some events occurring within the system. (e.g. an application launching, a printer event, ...)
- By constructing pointcuts. Opposite to RTTI structural changes, pointcuts control the dynamic part of the application. Pointcuts allow just intercepting the execution of constructions and methods (public or not).

In JAC, pointcuts are defined inside the Aspect components. Pointcuts are expressed as regular expressions. The information supplied to describe a pointcut is: (1) the *name* of the core object (any object hosted in a JAC container has a name); (2) the *class* name; (3) the *method* prototype, and (4) the name of the *container* where the object affected by the aspect is located. This is optional since the aspect, by default, is applied to all the hosts of the topology.

In order to facilitate the description of pointcuts, JAC defines a set of keywords that simplify pointcut writing and allow pointcuts to be independent from method names. Examples of these keywords are ALL, MODIFIERS, ACCESSORS, GETTERS, etc that represent all the methods of the matching objects that modify, access and get the value of the states of the object.

Aspect components in JAC can be configured by implementing a set of public methods in the Aspect component and by specifying the parameters of those methods in a separate Aspect Configuration File with .acc extension. Using this mechanism the

definition of pointcuts can be parameterised, increasing the reusability of aspects in different contexts.

Aspect Advice in JAC

Aspect behaviour in JAC is based on a dynamic wrapper. A dynamic wrapper is a regular stand-alone object with state attributes and methods. In addition, a dynamic wrapper can implement several methods that have special semantics.

Three different kinds of methods with special semantics are provided:

- Wrapping methods: Perform actions before and after regular object methods.
- Role methods: Extend regular objects interface (similar to *introductions* in AspectJ).
- Exception handlers: Handle exceptions that are raised by server objects in the object the wrapper is applied to.

These methods receive information about the intercepted joinpoint as a unique parameter (the *Interaction* parameter). This interaction contains the object that is wrapped by the wrapper, the currently called method and the parameters passed to this method. A base object can be wrapped by as many wrappers as needed.

Aspect Instantiation Mode

JAC provides two alternatives to instantiate aspects (wrappers). One of them is to create a unique instance of the wrapper shared by a set of base objects. The other one is two create a different instance of the wrapper for each base objects. Both wrapper instantiation modes can be seen in figure 6.

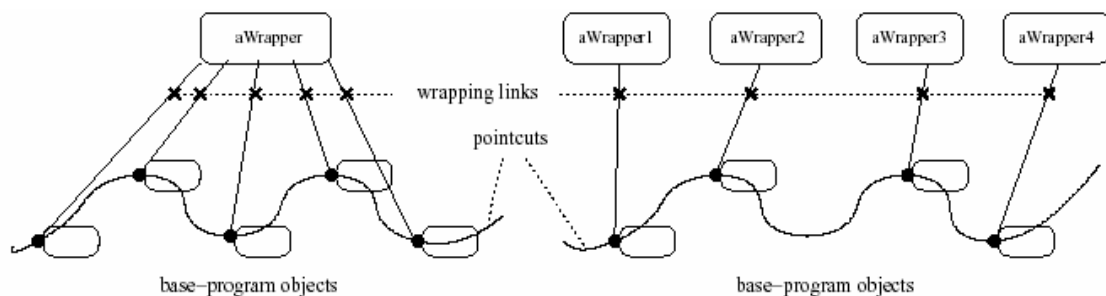


Figure 6. JAC wrapper instantiation mode (taken from [85])

The information about how to deploy wrappers is provided during the creation of pointcuts, when it is possible to pass a wrapper instance at the pointcut construction or let the pointcut deal with the wrapper construction. In the latter case, the value of the *one2one* flag must be passed to the pointcut. The default value is *false* to indicate that the wrapper is constructed only once and shared between all the base objects. If the value is *true* then there will be one wrapper instance per base object.

Predefined Aspects

At the aspect level, JAC provides a library of ready-to-use aspects. This library includes: GUI (Swing and Web), Session, Persistence (Hibernate is supported), Transaction, Deployment, Broadcasting, Authentication, Access Rights, Tracing, Cache, Integrity and Consistency. In order to reuse these aspects you only need to rewrite their '.acc' configuration files. So, in comparison with other approaches, JAC provides a high amount of built-in services, coded as aspects.

Of special interest are the aspects related to distribution, such as the deployment aspect. Given a set of hosts running a JAC daemon, the deployment aspect remotely install the application and the aspects. In addition, the deployment aspect also installs on each host the stubs which are needed for remote communications. Several other aspects related to distribution have been programmed: object replication, memory consistency, load-balancing.

3.7.2 Platform Infrastructure and Services

JACs final goal is to support distributed and dynamic AOP programming, which is achieved through its architecture called AODA (aspect-oriented distributed architecture). Its main purpose is to allow the natural and consistent cohabitation between aspects and distribution.

Infrastructure of JAC

Figure 7 shows the architecture of JAC and the flow of control of a JAC application. In JAC, functional classes (upper right side of figure 7) are modified by the JAC class loader at byte-code level (using BCEL), in order to make their instances wrappable. When an aspect is woven to a given application (upper left side of figure 7), the JAC system first reads the available aspect component configuration file ('.acc'). Then, the parser instantiates the corresponding aspect components and configures them by invoking a set of configuration methods. These invocations trigger the creation of pointcuts and the tagging of the classes with some meta-data. Finally, when a new instance of a base object is created, the Aspect-Component Manager automatically notifies all the registered aspects so that the pointcuts wrap its methods according to the aspect configuration.

This is a dynamic weaving mechanism and, therefore, any aspect component can be woven or unwoven at runtime. Its configuration file even can be parsed again while the application is running.

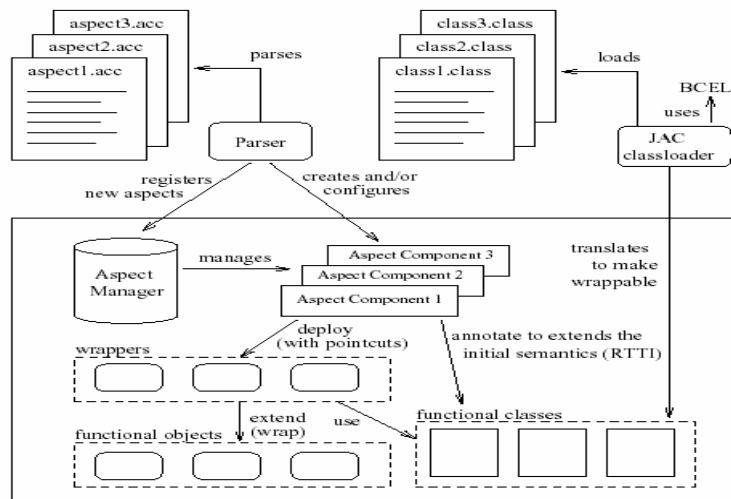


Figure 7. The JAC architecture (taken from [85])

JAC fully handles the distribution of aspects. The core distribution mechanism in JAC is based on the application of two kinds of aspect components: a deployment aspect, which is used to create a distributed-application, a set of distributed aspects that implement distributed protocols.

Services of JAC

As previously mentioned, JAC offers a set of predefined aspects and software developers only have to appropriately configure them in order to use them in their own applications. Some of these aspects offer common services usually offered by middleware platforms, such as persistence and transaction, distribution and monitoring.

Of special interest is the distribution aspect. Distribution with JAC has the following features: classes can be remotely uploaded on remote sites and objects can be remotely instantiated. Aspect component instances are replicated on hosts declared in the application *topology* and kept in synchronisation. The set of containers where an aspect is replicated is called an *aspect-space* (see figure 8). This means that each modification performed on one replica is automatically propagated on the others and, also, that all hosts access the same definition of aspect component and pointcuts.

When an aspect is distributed among different components the pointcut is also distributed. A distributed pointcut is a pointcut that is composed of joinpoints located in different hosts. As described before, the fourth parameter in the definition of a pointcut is used to define distributed pointcuts and it is a regular expression identifying the host names where containers are running.

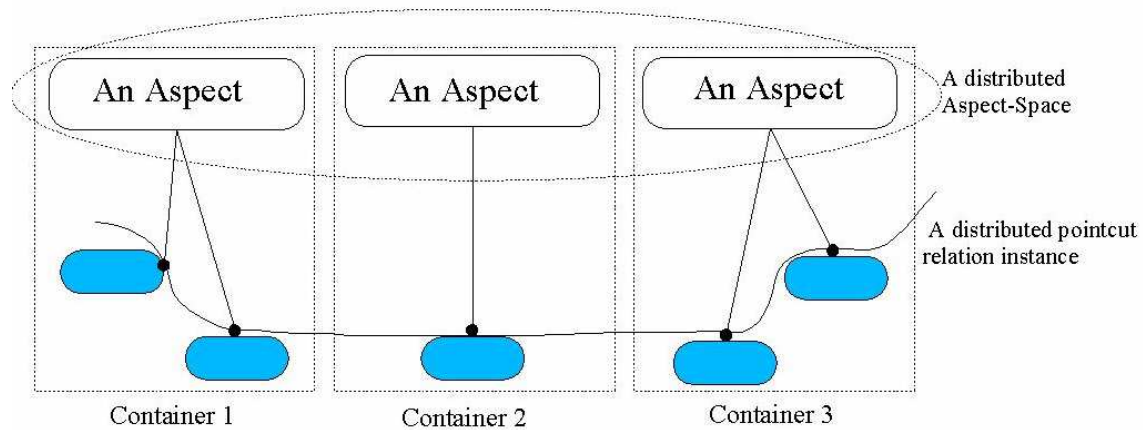


Figure 8. JAC distributed joinpoints and aspects

3.7.3 Current Status of Development

JAC is currently implemented in Java and released under version 0.12.1. In terms of middleware technology, JAC supports Java RMI and CORBA. The support for RMI is up to date and fully operational, whereas the support for CORBA works as a proof of concept prototype which needs to be upgraded. In addition, JAC also supports servlet (with the Jetty servlet engine) for programming GUIs.

The implemented aspects offered with JAC provides container-managed persistence, flexible clustering features (customisable broadcast, load-balancing, data-consistency, caching), instantaneously defined users, profiles management, access rights checking, authentication and distributed transactions with JOTM (Java Open Transaction Manager).

In addition to JAC, authors offer RAD, a Rapid Application Development approach to easily develop JAC applications. It includes the GUI aspect, which allows the programmer to develop Swing and Web applications very fast. This approach also includes a new notation for managing aspects in UML designs and the UMLAF IDE (UML Aspectual Factory), which will allow programmers to generate Java applications in a simply way, just by drawing boxes and by configuring existing aspect components.

3.7.4 Evaluation

According to the criteria used to compare middleware platforms for AOSD, JAC is a pure object-based framework. Entities which are been aspectised are plain old Java object. JAC performs dynamic weaving, and the joinpoint model is non invasive. The aspect programming model comes with the notion of aspect configuration: this notion allows an aspect to be configured and adapted to an application. The application dependent parameters can then be externalised from the aspect which is then more reusable.

With respect to *flexibility*, JAC offers the possibility of dynamically adding or removing aspects at runtime. The customisation is system-wide in the sense that the whole application (clients and servers) is impacted by the customisation.

With respect to *reliability* there is no a priori verification that the customisation could lead to inconsistencies (if for example, an old version of the aspect is currently in used). JAC leaves the management of this consistency to the aspect programmer, in that they can then decide if such a check is needed or not; alternatively they can skip it for performance reasons. The only mechanism provided by JAC regarding reliability is the enforcement of a system-wide global order for aspects. The purpose is to guarantee that two aspects will always be executed in the same order during all the entire application lifetime.

With respect to *performance*, JAC performs load-time adaptation of application classes. The purpose of this adaptation, performed with the BCEL byte-code engineering library, is to insert at the beginning of each method body an aspect management stub. This stub manages a list of woven aspects. The programmer can configure JAC to select the classes which need to be adapted (i.e. the classes where an aspect may be woven), thus limiting the runtime overhead to a given set of selected classes. A disk cache of adapted classes can be used to avoid unnecessary adaptations between two runs of the same program.

3.8 JBoss AOP⁶

JBoss AOP [87][88] is a Java-based aspect oriented framework that can be used in any programming environment or integrated in the JBoss application server (JBoss AS) [89]. JBoss AS 4.0 is a certified J2EE 1.4 application server. The description of JBoss AOP in this document focuses on the use of JBoss AOP integrated with the JBoss application server. JBoss AOP is a project of the JBoss Professional Open Source product suite.

3.8.1 Programming Model

Aspects in JBoss AOP are applied to Java object-oriented applications. Also, aspects and other constructions introduced by JBoss AOP are written in pure Java classes. These classes are bound to applications via XML documents or by using annotations.

In JBoss AOP the code of a core application can be affected by *aspects*, *interceptors*, *introductions* and *mixins*. An *aspect* is a plain Java class that encapsulates any number of advice, pointcut definitions, mixins, or any other JBoss AOP construct. An *interceptor* is an aspect that only has one advice method named *invoke()*. An *introduction* is used to force an existing class to implement an interface or to add an annotation to anything. Finally, a *mixin* class will implement a previously introduced interface.

Joinpoint Model and Configuration of Pointcuts

The joinpoints that JBoss AOP is able to intercept are field read and write, method and constructor invocations, points in which a method calls another method (“method call by method” joinpoint) or a constructor (“constructor call by method”), or points in which a constructor calls a method (“method call by constructor”) or another constructor (“constructor call by constructor”).

⁶ Web page, <http://www.jboss.com>

When a method or constructor calls another method or constructor, JBoss AOP distinguishes among the interception at the called side or the interception at the caller side. Additionally, joinpoints contained code within a particular call or a particular method or constructor can be intercepted. Two additional joinpoints are “has” and “has field”, which are additional requirements that a class that was intercepted by another pointcut must also be achieved. On the one hand, if a joinpoint is matched, its class must also have a constructor or method that matches the “has” expression. On the other hand, if a joinpoint is matched, its class must also have a field that matches the “hasfield” expression.

The joinpoint model in JBoss AOP is invasive, meaning that JBoss AOP can intercept these points in the execution of applications independently if they are public, protected or private.

In JBoss AOP pointcuts are described completely separate from both the aspects and the objects these aspects are applied to. Instead, they are configured using an external XML-based syntax. Pointcuts are expressed by means of regular expressions that can indicate information about the core class(es) affected by the aspects, specific method(s) inside that core class, the parameters of such methods, class fields etc. For each construction JBoss AOP provides a different pattern (*type pattern*, *method pattern*, *constructor pattern* and *field pattern*). Wildcards can be used to describe more general pointcuts. Pointcuts can be composed into boolean expressions using logical NOT, AND, OR and using parenthesis to group expressions.

Finally, JBoss AOP offers two ways of binding pointcuts and advice. One of them is by using XML and the other one uses Java 5.0 annotations. Using the XML <bind> element, JBoss AOP describes the binding between a pointcut and the aspects that must be applied on that pointcut. JBoss AOP resolves pointcut and advice bindings at runtime.

Aspect Advice in JBoss AOP

The advice of JBoss AOP aspects are encapsulated in aspects, which are plain Java classes, or in classes that implement the *org.jboss.aop.advice.Interceptor* interface. In an aspect class any number of advice methods can be defined. In the Interceptor interface only one method named *invoke(Invocation invocation)* will encapsulate the aspect advice. Using the *invocation* parameter the aspect has access to contextual information about the intercepted joinpoint and is used to drive the advice chain. It either calls the next advice in the chain, or it invokes the core code method or construction.

Aspect Instantiation Mode

During the configuration of aspects the JBoss AOP specifies the aspect scope, which defines the instant when an instance of the aspect must be created. An aspect in JBoss AOP can be created *per vm* (virtual machine), *per class*, *per instance*, *per joinpoint* and *per class joinpoint*.

If an aspect has a *per_vm* scope only one instance of the aspect class is created for the entire VM. A *per_class* scope indicates that only an instance of the aspect class is

allocated for a particular class. If an instance of the aspect must be created for each object instance, then the scope must be *per_instance*. Finally, a *per_joinpoint* scope indicates that an instance of an aspect is created for each joinpoint; there will be only an instance per class if the joinpoint is static and an instance per instance if the joinpoint is not static. If only one instance of the aspect should be created for each joinpoint independently of whether the joinpoint is static or not, then the scope must be *per_class_joinpoint*.

3.8.2 Platform Infrastructure and Services

JBoss AOP is integrated with the JBoss 4.0 application server, which is standard-compliant J2EE application server. Therefore, the execution of JBoss AOP applications relies on the architecture of a J2EE application server.

Infrastructure of JBoss

The infrastructure of the JBoss Application server is divided into four layers as shown in figure 9: (1) the *microkernel layer*, which delivers a lightweight component model that offers hot deployment and advanced class-loading features. It uses Java Management Extensions (JMX) [142]; (2) the *services layer*, which consists of a series of services such as transaction, messaging services, security services, etc. JBoss provides the mechanism to add new services not provided by the platform vendor. The software developer only needs to package the service as a *Service Archive*; (3) the *aspect layer*, which is based on the AOP model, and (4) the *application layer* which is where final applications live. Applications can use the container services directly or by using the AOP layer.

In order to deploy a JBoss AOP application in the JBoss application server, the application must be packaged. There are two different ways to package an application. The first one is to deploy an XML file in the “deploy/” directory of the JBoss application server with the signature **-aop.xml* together with the package that includes the application code. The second one is to include the XML file directly in the jar file that contains the application classes. The jar file must have the *.aop* extension and the *jboss-aop.xml* file must be contained in a META-INF directory of the jar.

By default, the JBoss application server will not do load-time byte-code manipulation of AOP files. The deployment user has to store, as part of the application server deployment files, a file containing some beans that deploy and manage the AOP framework. In addition the load-time transformation must be turned on by setting the *EnableTransformer* attribute to true.

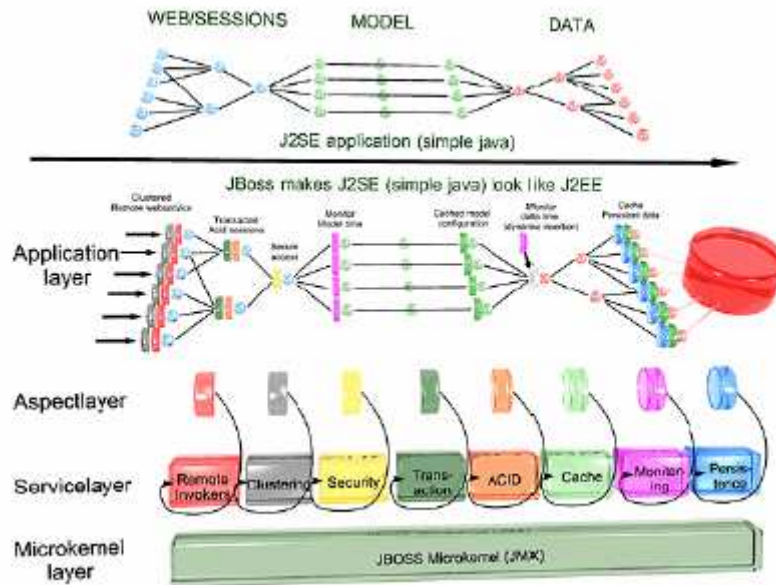


Figure 9. Architecture of the JBoss Application Server (taken from <http://www.jboss.org/products/jbossas/architecture>)

Services of JBoss and JBoss AOP

On the one hand, in the JBossMX 1.0 release, the JBoss microkernel layer provides different kinds of services such as standard, dynamic and model MBean instrumentation, MBean queries, timer, monitoring, relation and interceptor based invocation services, among others.

On the other hand, there are other services that can be downloadable as separate units and can be plugged into JBoss. Some of these services are EJB 3.0, clustering, remoting, IIOP, MQ, JTA, JCA, Tomcat, security, cache and IDE.

Finally, when JBoss AOP is integrated in the JBoss application server a pre-packaged set of aspects can be applied via annotations, pointcut expressions, or dynamically at runtime. Some of them are caching, asynchronous communication, transactions, security, remoting, a read-write lock aspect, a design by contract aspect, a thread-based aspect, a GOF pattern (Gang of Four), among others.

3.8.3 Current State of Development

JBoss AOP is one of the several services that can be integrated within the JBoss application server. As mentioned before, another important set of services is also provided and can be integrated into JBoss as plug-in modules.

In addition, different tools exist to support the construction of JBoss AOP applications. One of them is the JBoss AOP IDE that is an Eclipse plug-in that helps you to define interceptors via a graphical interface and to run applications from within Eclipse. Other tools are the debugging and analysis tool, which are available if applications are run within the JBoss application server. Concretely, the JBoss application server offers a GUI management console that shows information about instrumented classes.

3.8.4 Evaluation

The JBoss server and container are completely implemented using component-based plug-ins. The modularisation effort is supported by the use of JMX, the Java Management Extension API. Using JMX, JBoss results in a very extensible and modifiable middleware platform. For instance, this mechanism is used in JBoss to support extension with respect to binding types. On the one hand, in order to be J2EE compliant, JBoss AS 4.0 implements JAX-RPC (Java API for XML for Remote Procedure Call) to support J2EE Web Services. It also implements JMS 1.1 (Java Messaging Service). On the other hand, new binding types can be added to the microkernel by using plug-in modules. Examples of available modules that add new binding types to JBoss AS are the JBoss Remoting and the JBoss IIOP APIs. The JBoss Remoting API provides support for synchronous and asynchronous remote calls, and push and pull call-backs. The JBoss IIOP API supports CORBA/IIOP access to turn JBoss into a CORBA application server. In addition, new modules with new binding types may be incorporated into the JBoss AS.

The micro-kernel architecture permits JBoss to scale from embedded industrial environments to large scale environments, with advanced integrated remote installation features. The JMX feature provides technology for monitoring and managing devices, applications, and networks. This standard is suited for integrating into legacy systems, as well as implementing new monitoring and management solutions.

Once again because of the JMX architecture, JBoss AS provides higher flexibility than similar container-based technologies such as CCM/CORBA or EJB/J2EE regarding the number of container-provided services. Services can be easily added or removed based on the specific needs of software developers. They can also build their own services and deploy them as SARs (Service Archives) within the JBoss AS. Each SAR is individually hot-deployable, making it very easy and flexible to extend JBoss. Additionally, the flexibility offered by the open set of container-provided services offered by JBoss is increased by using JBoss AOP, that allows easily and transparently weaving in the behaviour provided by services into any objects.

The micro-kernel approach leads to a high performance application server. Based on their own internal benchmark tests, JBoss users have found that JBoss offers improved performance and superior server utilisation over other leading J2EE application servers. By using the Monitoring JBoss tool it is possible for automatic monitoring, notification and correction of performance, availability and usage statistics of JBoss Servers, providing detailed information on every component or service deployed on JBoss.

Finally, according to the criteria used to compare middleware platforms for AOSD, JBoss AOP is an invasive approach developed to apply aspects to plain Java objects. Since it is integrated into a J2EE-compliant application server, aspects can also be applied to EJB components. The JBoss AOP used as an application layer on top of the JBoss AS is very flexible regarding the variety of aspect instantiation modes that can be provided as part of the deployment information. JBoss AOP promotes aspect reuse avoiding explicit pointcuts and binding information in the aspect definition, which are described in external XML files. The addition/removal of aspects can be carried out

both at compile and load-time. Though weaving is performed at load time by byte-code transformation, there is a Dynamic AOP API that allows adding/removing advices and interceptors on any joinpoint that was aspectised during compilation or load time. This makes final applications highly adaptable and extensible.

3.9 Lasagne⁷

Lasagne [90][91] is an aspect-oriented middleware for context-sensitive and dynamic customisation of distributed services. It is used for the construction of customisable middleware and distributed services. Lasagne is currently developed by the Distributed Systems and Computer Networks (DistriNet) group at the Katholieke Universiteit Leuven (KUL), and the Maersk Mc-Kinney Møller Institute at the Southern University of Denmark.

3.9.1 Programming model

Lasagne defines a platform-independent architecture for dynamic customisation of component-based distributed systems. Lasagne defines its own component and aspect (named *collaboration* in Lasagne) programming model, not being an extension of any standard component model. The Lasagne programming model is based on the definition of *collaborations*, *wrappers*, *component descriptors*, and *interceptors*.

In Lasagne a distributed service is structured consisting of a minimal functional core, implemented as a component-based system, and a set of potential *collaborations* that can be selectively integrated within this core functionality. Therefore, aspects are modelled as *collaborations*, which consist of several *wrappers*, each one to be wrapped around a different point in the core system. These collaborations can be both new functional services to incorporate new functionality to an existing application, and non-functional services.

A *component* is a coherent unit of one or more classes, with an explicit *component type*. Similar to other component models, components in Lasagne are described by the set of interfaces they provide (their *service interfaces*) and the set of interfaces they depend on (their *dependencies*). This information must be known by Lasagne *wrappers*, which have to specialise the interfaces of the components they wrap.

An interaction between a client system and the core system (i.e. a client request) initiates in Lasagne a (distributed) message flow between the component instances of the core system. Multiple client requests may be concurrently processed by the core system.

The Lasagne model is founded on four conceptual features: (1) the extension of the object-oriented programming model with the notion of *aggregate object identity* that unites and hides the separate object identities of the component instance and its wrapper instances; (2) the introduction of the notion of *feature identifier*, that uniquely identifies a collaboration; (3) the external definition of the collaboration selection logic in a *composition policy* that travels with the distributed message flow, and (4) the incremental definition of the composition policy at runtime using *interceptors*. An interceptor intercepts incoming and outgoing messages of a specific

⁷ Web page, <http://www.cs.kuleuven.ac.be/~eddy/lasagne.html>

component instance and may update their associated composition policy by attaching/discarding feature identifiers.

Joinpoint Model and Configuration of Pointcuts

According to the philosophy of CBSD, Lasagne is based on a non-invasive joinpoint model, where aspects only intercept the incoming and outgoing component messages described in component interfaces and never the private state of components or private methods. When a joinpoint is intercepted Lasagne offers to the aspects context information about the joinpoint.

In Lasagne, the composition logic (or aspect pointcuts) responsible for integrating collaborations into the core components is completely separate from the code of the components, and of the collaborations as well, increasing their reuse in different contexts. This information is specified in *component descriptors* files.

Component descriptor files describe the provided and required interface of components, together with the list of wrappers that can be wrapped around the component. These files also provide information regarding the order in which the wrappers must be executed. A final item in component descriptor files are *method bindings*, which refine the definition of pointcuts by indicating the specific methods sent or received by a component that should be intercepted by a wrapper.

Aspect Advice

As mentioned before, the advice of a Lasagne aspect is described as a collaboration, whereby each collaboration is a set of one or more wrappers. The wrapper programming model of Lasagne combines the benefits of both the Decorator and the Role Object design pattern as well as meta-programming facilities. On the one hand, wrappers are programmed similarly to a component-oriented decorator, which attaches additional state and refined behaviour to a dynamically bound inner component instance. This decorator-based programming style is similar to around advice where method bindings are specified implicitly. Lasagne can also attach new service interfaces to a component instance using the Role Object design pattern, which is similar to inter-type declarations. On the other hand, Lasagne also supports reusable, multi-signature around advice in the form of *meta-wrappers*. As opposed to decorator-style wrappers, meta-wrappers define advice in a highly reusable and versatile manner by means of message interception. A disadvantage of using meta-wrappers, however, is that one must specify explicit method bindings in the component descriptor files and one loses the type checking capabilities that one gets for free in a decorator-based programming style.

Wrappers consist of two different blocks, a *wrapper specification* and a *wrapper implementation*. First, in the wrapper specification one binds the wrapper with a collaboration by declaring the feature identifier which it belongs to, one specifies the expected core component interfaces (following the Decorator pattern) and the addition of new service interfaces (according to the *Role Object* pattern). Secondly, the wrapper implementations are Java classes that (re)implement the interface of the component they decorate, adding the advice code before and after the original

component code. The core component is referenced using the *inner* variable, which is bound at runtime.

Wrappers also have access to the *contextual properties* of the ongoing collaboration. These properties provide wrapper information about the calling client. Wrappers in Lasagne can avoid direct interaction among themselves using these contextual properties. One wrapper can attach context information to the distributed message flow and this information can be later restored by another wrapper.

Aspect Instantiation Mode

In Lasagne wrappers are instantiated per component instance (instance-level instead of class-level). The behaviour of a collaboration can be consistently turned on and off on per distributed message flow basis, making the selection of collaborations context-sensitive and dynamic.

During the deployment of the application, the *component descriptor* specifies which wrappers must be applied to core components, and specifies partial order constraints between wrappers, defining an application chain of wrappers. The application of these wrappers occurs at deployment time and on a per component basis.

3.9.2 Platform Infrastructure and Services

Lasagne is just a specification, like CORBA, that you must implement in order to use its features. The main purpose of Lasagne is to develop a runtime aspect weaving mechanism that can be integrated on top of existing object-oriented and aspect-oriented programming languages and middleware platforms.

Infrastructure of Lasagne

In order to extend an existing programming language or middleware platform with Lasagne, the general overview of the Lasagne runtime system shown in figure 10, must be able to be implemented in that language or platform.

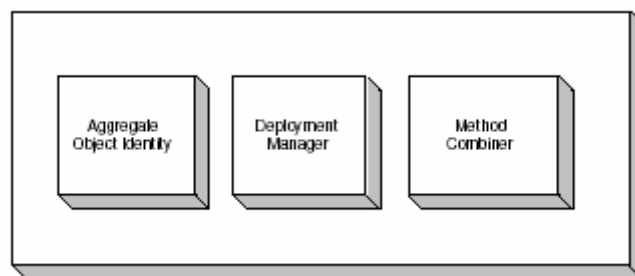


Figure 10. High level functional decomposition of the runtime system (taken from [90])

1. *Aggregate Object Identity*: This module is responsible for presenting a core component instance and its wrapper instances as having the same aggregate object

identity. This is a requirement of Lasagne to be able to implement the dynamic wrapping model.

2. *Deployment Manager*: This module manages the binding between selectors (method signature that uniquely identifies the invoked operation) and methods, and the order in which methods, that are bound to the same selector, must be combined. It is composed of a method map, an object adapter map and a component type manager. A method map has the structure of a table with an entry for each selector supported by the aggregate object. An object map knows how to create an instance of a wrapper on demand and how to invoke methods on it. Finally, the component type manager manages type information about the aggregate object.
3. *Method Combiner*: For every message received, the method combiner of the receiver object will selectively combine the methods that are bound to the selector of that message. The composition policy determines to which methods the message will be dispatched.

The two most important runtime mechanisms for dynamic weaving in Lasagne are the method combiner and the deployment manager. Figure 11 shows how these two modules interact among each other to perform the dynamic weaving. The figure shows a situation in which three different collaborations have already been deployed for a core component. Step 1 shows that a new message was sent to the object. The method combiner must dispatch the corresponding message to the appropriate combination of methods (step 2). In order to do that, first it determines the method list that matches with the selector of the message (step 3) and second it has to execute those methods in the list, which are selected using the information in the composition policy of the message (steps 4,5,6). Before executing a message the method combiner checks that the wrapper has already been constructed (step 4.a). If during the execution of a method other operations are invoked, a new message is created for each invocation (step 4.b) and the composition policy is copied into them (step 4.c).

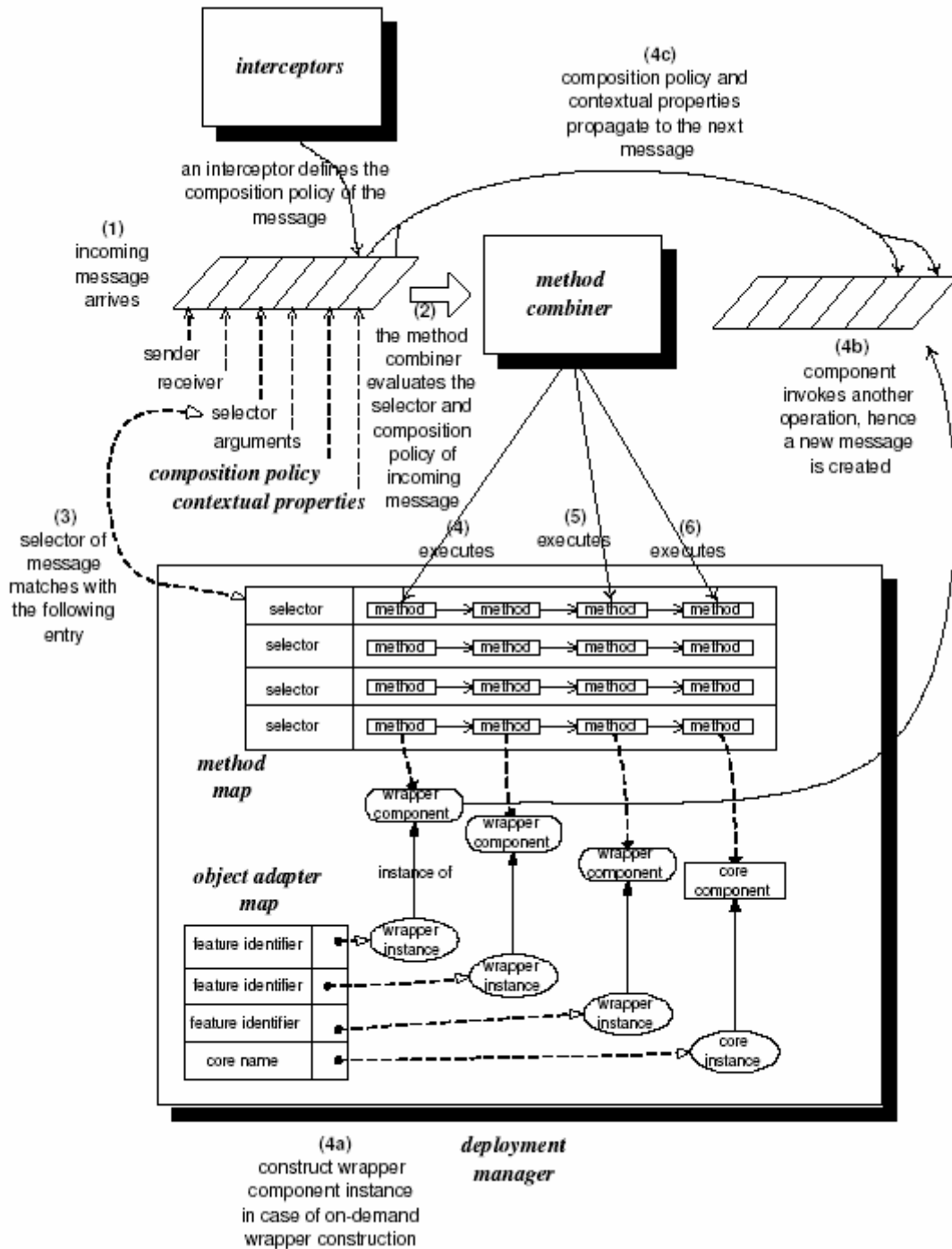


Figure 11. Runtime weaving mechanism in Lasagne (taken from [90])

3.9.3 Current State of Development

Lasagne defines a platform-independent architecture. It can be implemented on top of a language or middleware platform with an open implementation. Currently, there are implementations of Lasagne on top of the concurrent object-oriented language Correlate [143] (using Correlate's MetaObject Protocol) and on top of Java (using load-time reflection).

Correlate is a concurrent object-oriented language with an execution environment that makes it easier to develop distributed applications. Correlate classes are compiled to normal Java classes, so Correlate runs on any JVM platform. Using Correlate the Lasagne runtime system has been implemented as a modular plug-in. A feature of Correlate that facilitates the development of distributed applications is that the execution environment it provides has its own middleware layer.

Lasagne/J is a Java extension that supports the Lasagne semantics and offers higher-level object-oriented language constructs. Current and future research directions for Lasagne/J include the development of IDE support for visualisation of relationships between generic wrappers and classes, investigation of Lasagne/J's effect on the application development lifecycle, and continuous experiments with the language's features.

3.9.4 Evaluation

With respect to the *criteria used to compare middleware platforms for AOSD*, Lasagne can be classified as follows. Lasagne is a non-invasive approach developed to apply aspects to component-based applications. Its component model is not an extension of a standard component model. Weaving of aspects is performed at runtime using message interception. Lasagne allows to program advice either in a very application-specific, non-reusable way – using a decorator based programming style, or in a very reusable and versatile way – using a meta-level programming style. The former is generally more suitable for aspectualising functional services, whereas the latter is generally more suitable for aspectualising non-functional services.

With respect to *flexibility*, Lasagne offers a high adaptivity with respect to dynamic and client-specific selection of aspects; the selection of which aspects to execute can be changed at runtime on a per client request basis. The specification of which component methods must be advised by an aspect (i.e. the pointcut specification) is performed at two different times: in the case of decorator-based wrappers, pointcuts are specified implicitly when programming the wrapper; in the case of meta-wrappers, pointcuts are specified during deployment time using the mechanism of method bindings.

With respect to *performance*, the existing Lasagne implementations induce a high performance overhead. This is due to the fact that there is a message indirection every time control flow crosses aspect boundaries. This message indirection can probably be eliminated by using implementation techniques that are situated at the virtual machine level and reusing existing techniques for an efficient implementation of dynamic dispatch.

With respect to *reliability*, Lasagne focuses mainly at protecting the system against inconsistencies that arise from client-specific selection of aspects. We distinguish between three levels of consistency management: global behaviour consistency, global state consistency and system integrity.

- Global behaviour consistency. This means that the execution of an aspect happens in an all or nothing fashion. When a client dynamically selects a combination of aspects for processing its client requests, Lasagne activates this combination of

aspects atomically over the entire system. As such no behavioural inconsistencies can arise. In fact, the novelty of Lasagne is that it enables to consistently turn aspect behaviour on and off.

- Global state consistency. This means that different aspects modify and observe shared state in a mutual consistent manner. Lasagne does not provide runtime support for this, but instead it specifies programming guidelines for this purpose. More specifically, Lasagne requires that all shared state is completely separated from the wrappers and instead centralised into the core components. Wrappers do not define shared state of their own but are developed as a set of methods that operate on the shared state that is defined by the core component. This property ensures that modifications made via calls to the core component are visible to other wrapper components. In order for this to work, however, it is important that the design of the core component is complete in the sense that any useful customisation can be defined in terms of the provided service interface of the core component. This requires careful design and programming of core components.
- System integrity. This means that clients may not be able to inject malicious aspect code into the system or select dangerous combinations of aspects. Lasagne provides protection against this kind of inconsistencies by means of deployment-specific interceptors that check the composition policy of every incoming client request.

3.10 PRISMA

PRISMA [92][93][94] is a conceptual model which enables the description of distributed software architecture by combining CBSE and AOSD. The PRISMA model has been implemented in C# .Net. PRISMA has been developed by the Department of Information Systems and Computation at the Polytechnic University of Valencia.

3.10.1 Programming Model

PRISMA is a model to define architectures of complex software systems. The PRISMA architectural model consists of different types of elements, which are *interfaces*, *aspects*, *components*, *connectors* and *systems*.

An *interface* provides a set of services. An *aspect* is the join of a set of interfaces and the semantics specification of its structure and behaviour (state changes, triggers, protocols, etc.). PRISMA considers different kinds of aspects: functional, distribution, coordination, quality, etc, which are defined in an orthogonal way. In PRISMA there is not the concept of base program that is advised by aspects, since functionality is considered as another aspect.

A *component* is an element that is composed of a set of *aspects* and of a set of in-ports and out-ports. In the PRISMA model components interact with other architectural elements by their ports. Therefore, PRISMA offers a different definition of *aspect*, where an aspect is the complete definition of the structure and behaviour of a system from a particular viewpoint or concern. Notice that an aspect may be any kind of viewpoint, including the functional one.

A *connector* acts as the coordinator between the elements in the model. Similar to components, connectors are composed by aspects (coordination, functional, distribution, quality, etc) and a set of in-roles and out-roles, whose type is a specific interface. Connectors link and synchronise components, systems and other connectors by means of their roles.

Finally, a *system* in PRISMA is a component that includes a set of connectors, components and other systems correctly connected. In PRISMA all these elements and the relationships among them are described by means of a *component definition language* (CDL), which has as first-order entities: interfaces, aspects, components and connectors. By considering interfaces and aspects as first-order entities, the aim of PRISMA is to increase the reusability since an interface may be used by several aspects and an aspect may be used by several elements (components and connectors).

Joinpoint Model and Configuration of Pointcuts

The joinpoint model in PRISMA is defined in terms of the interfaces of aspects, the in-ports and out-ports of components and the in-role and out-role of connectors. Basically, each kind of aspect defines a set of interfaces. Then, in order to construct a component as the composition of a set of aspects, the type of in-ports and out-ports are of a specific interface. Additionally, in order to construct a connector as the composition of a set of aspects, the type of in-roles and out-roles are from a specific interface. The joinpoint model can be seen as non-invasive, since the connections are always established in terms of the interfaces of the different elements in the system.

The composition between the different elements in PRISMA is described by means of two different languages. The CDL is used to describe all the elements in the model: interfaces, aspects, components and connectors. As part of the description of a component the *Weaving* clause indicates how the different aspects that set up a component are weaved among them (pointcuts). The Configuration Language is used to describe the architecture of a specific application in terms of the elements previously described with the component description language.

Aspect Advice in PRISMA

During the description of pointcuts, in addition to the information about which aspect is composed with another one, PRISMA indicates whether the sequence of executions is before, after or around.

Aspect Instantiation Mode

In PRISMA aspects are instantiated in a *per-component* basis if they are part of the definition of a component or in a *per-connector* basis if they are part of the definition of a connector.

Predefined Aspects

PRISMA provides a set of predefined aspects, including the following: functional, distribution, replication, context-awareness, quality, coordination and evolution. This number is not limited as new aspects can be defined.

One of the most interesting aspects in a PRISMA application is the distribution aspect. In PRISMA the distribution aspect have to be added to the set of aspects types of a conceptual model in order to enable the specification of software architectures of distributed systems. The distribution aspect specifies the features and strategies that manage the dynamic location of instances of architectural elements in a software architecture. The distribution aspect deals with all the properties related to distribution and changes in location. Each architectural element with a distribution aspect must have a location.

3.10.2 Platform Infrastructure and Services

PRISMA is an architectural model than can be used to describe the architecture of software applications based on components and aspects. Applications designed with PRISMA have to be implemented using different kinds of object-oriented and/or aspect-oriented languages. Until now, PRISMA has been implemented in .NET. In order to do that a middleware platform has been defined on top of .NET.

Infrastructure of PRISMA

PRISMA has been implemented using the C# .Net and the .Net Common Language Runtime (CLR). It uses .Net Remoting to offer distributed communication and mobility.

In order to implement PRISMA applications, an abstract middleware above the .Net CLR has been developed (see figure 12). The middleware consists of a set of constructs (classes, structures, etc.) implemented in C# .Net primitives to offer extra functionalities and characteristics that are required by PRISMA and that are not directly offered by .Net, such as the aspect-oriented approach and the mobility of components.

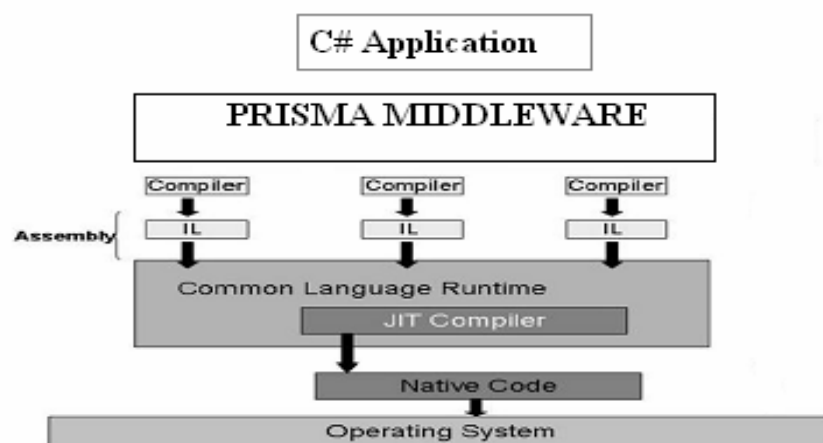


Figure 12. PRISMA architecture (taken from [92])

The base classes and structures in the PRISMA middleware are *Aspects*, *Components*, *Attachments*, *Binding* and *Middleware* to implement all the elements of the PRISMA

model. In addition, the middleware platform contains a set of classes necessary to solve exceptions which may occur at runtime.

The PRISMA middleware has to run on each node where a PRISMA application needs to be executed. Therefore, the middleware layer provides a set of classes that enable the distributed communication among the different middleware layers of the PRISMA architectural model.

The weaving mechanism defined by PRISMA is dynamic at runtime. When the execution of the application is initiated those aspects that are part of a component are added to that component as defined in the description of the application architecture. This list of aspects can be adapted at runtime since aspects are managed by a dynamic list of delegates that allow adding and removing aspects at runtime.

Services of PRISMA

The main service offered by the middleware layer developed on top of .NET is to provide support to develop aspect-oriented applications conforming to the PRISMA model in .NET. In addition it provides distributed communication and mobility using .NET Remoting.

3.10.3 Current Status of Development

The current implementation of the middleware platform for PRISMA in .NET contains the base constructs to enable the execution of distributed PRISMA applications in .NET.

Future work includes the building of a model compiler to perform automatic code generation of distributed applications from the PRISMA specifications. This model compiler will be part of the PRISMA framework which will have tools to support the modelling of the PRISMA specification in a graphical notation.

3.10.4 Evaluation

With respect to the *criteria used to compare middleware platforms for AOSD* PRISMA defines a new component and connector based model, where components, connectors and the connections among them are described at the architectural level. Both components and aspects are the composition of a set of aspects, including a functional aspect. The joinpoint model in PRISMA is non-invasive since the weaving between aspects is defined in terms of their interfaces. All the elements in the architecture of an application are defined using a component definition declarative language. In this language, the interfaces provided by aspects and the aspects themselves are first-class entities being possible to reuse them in different contexts. Furthermore, pointcuts are not defined as part of aspects, but as part of the components increasing the reusability of aspects. Application adaptability is supported since aspects can be added and removed from components at runtime.

With respect to *flexibility* in PRISMA the flexibility of final applications is supported by using two main mechanisms. The first one gives the possibility of adding new

components/connectors to an application at runtime. The second one gives possibility to modify the number of aspects that constitutes a component or a connector also at runtime, which can be added/removed from the elements in the model (components, connectors) during the application execution.

With respect to *reliability* the connections between aspects are defined in PRISMA at the architectural level using a formal language. This language allows to validate the weaving information provided during the description of the application architecture and to automatically generate the application code from the information captured in the models. This assures the consistency between the validated architecture and the generated code.

With respect to *performance* there are no available reports from authors. Basically, the current implementation of PRISMA relies on the use of the .NET platform and its distributed communication mechanism (.NET Remoting).

3.11 PROSE/MIDAS⁸

The PROSE (PROgrammable extenSions of sErVICES) system [95][96] is a dynamic weaving tool that allows inserting and withdrawing aspects to and from running applications. MIDAS (MIDleware Adaptive Services) [97][98] is a system based on PROSE that allows applications to self-organise into spontaneous information systems, but without relying on a fixed infrastructure. Prose/Midas is developed by the Department of Computer Science in the Swiss Federal Institute of Technology Zürich (ETH Zürich).

3.11.1 Programming Model

MIDAS and Prose are both based on the Java language. MIDAS is a middleware layer for adaptive services, and Prose is the language providing the facilities required by MIDAS.

MIDAS is based on the *spontaneous container* concept. A spontaneous container is a container that adapts computer appliances to the environment where they are being executed. It works dynamically for entire service communities, which are built dynamically, using dynamic service discovery. For instance, consider a node “A” joining a service community. In a service discovery and brokerage environment, “A” will be discovered by other services and will discover by itself the services it needs. But there is no guarantee that “A”’s state is recoverable, robust, and secure. If the service community we are talking about is a *spontaneous container*, then all the needed features are added dynamically to “A”. A will work now using enterprise-strong security and robustness, without losing the option of leaving the community or discovering and using the new services.

In MIDAS a service community consists of fixed or mobile computer nodes (see figure 13). Every node (green box) exports one or several services (grey boxes), as shown in the figure. The figure depicts the situation in which one new node (marked with a large white arrow) joins the MIDAS community. The MIDAS community

⁸ Web page, <http://prose.ethz.ch/Wiki.jsp>

contains a special computer or node called JEB (JEB stands for Java Extension Base) which discovers the newcomer. It then sends two extensions to the newcomer. The extensions are based on Prose. To be able to receive extensions, the newcomer has a special component called JEM (JEM stands for Java Extension Manager), also based on Prose, and can receive extensions from other nodes and apply them. After the newcomer has received the two extensions, the JEM activates the extensions.

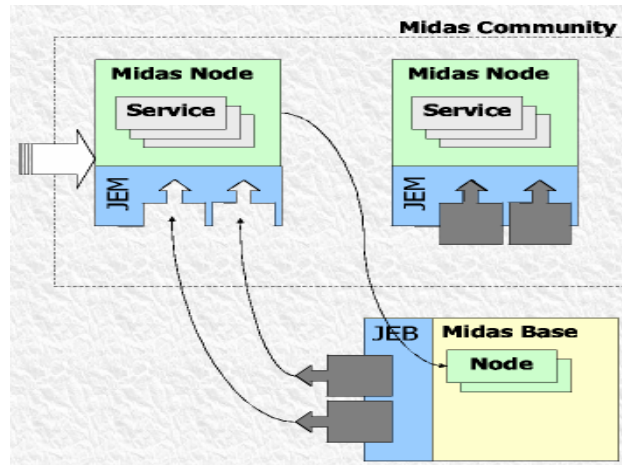


Figure 13. MIDAS service community (taken from <http://prose.ethz.ch/Wiki.jsp?page=AboutMidas>)

Aspects in Prose/MIDAS are written using the aspectual language Prose. Prose aspects are regular JAVA objects that can be sent to and be received from computers on the network. Aspects in PROSE are applied over simple objects that are dynamically bound and unbound into an object-oriented application at runtime.

Joinpoint Model and Configuration of Pointcuts

PROSE aspects can be applied before and after method executions, field accesses and modifications, within the code of methods, class load, etc. Therefore, PROSE aspects can intercept points that are part of the internal behaviour of objects defining an invasive joinpoint model.

Pointcuts are written inside *crosscuts* classes, which are classes that must inherit from one of the following abstract classes: *MethodCut*, *ThrowCut*, *GetCut*, and *SetCut*. These classes include not only the definition of pointcuts, but also the aspect advice. Pointcuts are defined in the *pointcut()* method of Crosscut classes. In addition, pointcuts can be refined by using the arguments of the advice methods (crosscut method executions).

A wide collection of classes, keywords and logical operators for filtering pointcuts inside crosscut method executions is provided in Prose. In addition, all these classes provide the *thisJoinPoint()* method to obtain contextual information about the intercepted joinpoint.

Aspect Advice in PROSE/MIDAS

In order to implement an Aspect in PROSE software developers have to extend the *Aspect* class and defines different attributes of type *Crosscut* to implement the different advice contained in those aspects.

Aspect advice in PROSE is written as a method inside Crosscut classes. Each of these crosscut classes executes the advice method for specific kinds of join-points: *MethodCut* captures boundaries of method executions; *ThrowCut* captures exception throws; *SetCut* captures field modifications, while *GetCut* captures field accesses.

The name of the methods depends on the type of the class. If the crosscut class is *MethodCut*, the `METHOD_ARGS()` method encapsulates the aspect advice that will be executed before or after executing the method. If the crosscut class is *SetCut* or *GetCut*, the aspect advice is encapsulated in the `SET_ARGS()` method or in the `GET_ARGS()` method, respectively. Finally, if the class is *ThrowCut* the name of the method is `THROW_ARGS()`.

These methods can have different kinds of parameters that, as mentioned before, allow refining the definition of pointcuts. Therefore, it is possible to parameterise pointcuts using the parameters of such methods.

Aspect Instantiation Mode

Execution of pointcuts is ordered by a priority that can be specified in the aspect code. Aspects can be deployed per class, indicating that the same aspect is shared by all objects belonging to that class, or per instance, each object having its own aspect instance.

3.11.2 Platform Infrastructure and Services

In this section we are going to distinguish between the architecture of MIDAS, where PROSE is integrated, and the architecture of the dynamic weaving mechanism in PROSE.

Infrastructure of PROSE/MIDAS

MIDAS architecture is shown in figure 14.

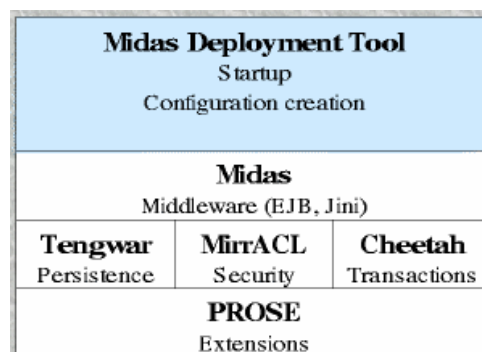


Figure 14. MIDAS architecture (taken from <http://prose.ethz.ch/Wiki.jsp?page=AboutMidas>)

It is composed by the following components:

1. The *Midas Deployment Tool* (at the top), is an XML-based tool that allows the configuration of entire service communities. This tool is present once in every network and generates configuration objects, which can be easily transformed into Prose extensions.
2. The *Midas Middleware* is a module that must be present in every node willing to become a member of a spontaneous container. Midas middleware receives extensions and passes them to Prose.
3. *Prose* is used to achieve run-time adaptation. Every node in a container must have Prose activated. Prose uses several sub-components. Using these sub-components, enterprise features can be added to arbitrary services. This enterprise-service functionality is based on Cheetah, MirrACL, and Tengwar.
4. *Cheetah* is a system that allows transactional correctness for arbitrary and dynamic service hierarchies. Cheetah has become a commercial product, available at Atomikos.
5. *Tengwar* is an object-relational mapper (ORM). It is extremely useful for mapping object states to relational tables in a simple and efficient way. Unlike many other ORMs, it is adapted for working with aspect-oriented features and allows in a certain sense orthogonal persistence of Java objects.
6. *MirrACL* is a Jini-based system for distributing and creating network-specific identities (public/private keys). This allows nodes that do not know anything about authorisation, authentication and access control to become security-aware.

In PROSE the weaving process is dynamic. Aspects can be inserted and removed through an *Aspect Manager* in a transactional way. Weaving works on dynamically (late) loaded classes. Insertion and removal in remote JVM's is allowed.

Weaving mechanism is based on the schema shown in figure 15. A two layer weaver is proposed. The *AOP Engine* accepts aspects and generates joint point requests, which are properly activated invoking methods of the *Execution Monitor*, which is integrated with the JVM. When a joint point is reached, the AOP Engine is notified by the execution monitor and the corresponding advice is executed. Execution monitor is responsible for addressing performance under normal operations and efficient advice execution, while AOP Engine must address secure and atomic weaving, and hide platform specific features improving flexibility.

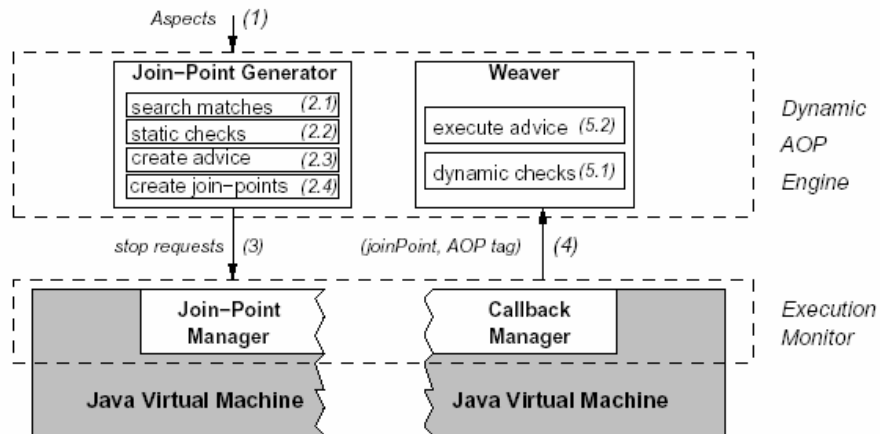


Figure 15. PROSE architecture (taken from [96])

Once an aspect has been inserted in the JVM, any occurrence of the events of interest results in the execution of the corresponding aspect advice. If an aspect is withdrawn from the JVM, the aspect code is discarded and the corresponding interception(s) will no longer take place.

Services of PROSE/MIDAS

PROSE does not offer a predefined set of aspects ready to be used. Instead, PROSE predefines a number of frequently used *pointcutters* that can help to easily developed new aspects. Pointcutters allow the execution of the advice depending on various properties of the reached execution point, and achieve this goal by implementing a filtering mechanism on join-points. This filters can act as run-time (run-time filtering) or at insertion-time (insertion-time filtering).

3.11.3 Current Status of Development

MIDAS is currently released under version 0.2, and Prose under version 1.2.1. PROSE works under Linux, Windows, Mac OsX and Solaris.

There are three different prototypes of PROSE: one based on the debugger interface of the JVM ("JVMDI-based weaver") and the other two based on JIT compiler technology ("hook weaver" and "stub and advice weavers").

Aspects can be added to a running PROSE application by directly using the command line or by using *wbprose* a GUI tool they have developed to insert/delete aspects from local and remote JVM where PROSE is running. This tool has also been developed as an Eclipse plug-in.

3.11.4 Evaluation

Regarding to the criteria exposed for middleware in Section 2, PROSE/MIDAS does not have special features furthermore than exposed in EJB/Jini, which is the middleware layer over which MIDAS is constructed. Then, binding management is absolutely dependant of EJB/Jini binding management, having its same advantages and disadvantages. Some kind of resource adaptation and resource management

policies could be achieved using properly spontaneous containers. Spontaneous containers were designed to provide dynamic and context-sensitive customisation. However, easy resource adaptation and extension of resource management policies is not provided, being in many cases non-intuitive to write the necessary aspects for achieve the desired goals.

Some degree of flexibility is achieved in PROSE allowing the definition of new joinpoints and some degree of customisation of native layer. PROSE/MIDAS offers a dynamic aspect-oriented approach, which is always optimal in order to adapt applications at run-time. In addition, the two layer weaving architecture allows the reuse of the execution monitor or the AOP Engine independently, according to different purposes. For example, for an adaptation of Prose to a more efficient JVM there will only be a need to rewrite some code from the execution monitor, without rewriting anything from the AOP Engine.

Any customisation of the MIDAS middleware platform has to be done through aspects. Aspects in Prose are not easily reusable or easy to write for AOP programmers because they contain several concepts tangled. Pointcuts are defined among pointcut declaration and advice methods inheriting from several classes and writing several abstract methods. Control over inconsistencies introduce by addition of new aspects isn't provided.

Binding management is dependent of middleware layer, which is EJB/Jini. Pointcuts and advices are coded together, so any kind of binding between pointcuts and advices isn't necessary.

Regarding performance, the focus of PROSE/MIDAS is towards highly optimised dynamic weaving through a JIT compiler. Test of execution monitor shows a 5%-10% overhead when no aspects have to be woven, and when aspects are woven, the cost of advice invocations is similar to an "invokeinterface" call. Dynamic aspect execution is 1.5%-5% slower than in static approaches like AspectJ. When AOP Engine and execution monitor are measured together, the cost of executing an empty advice in Prose is 2.5% to 8.5% higher than in AspectJ.

In relationship with AOSD criteria, PROSE/MIDAS is designed for Java, and to be used over EJB or Jini. They offer a dynamic weaving mechanism based on a JIT-compiler, focused on performance. Joint point model is invasive, allowing write complex conditions to intercept different points during the application execution. Aspect reusability is low, being pointcut declaration and advice code mixed, and being pointcut declaration spread over several places, making difficult the reuse of aspects between applications. Spontaneous container in MIDAS provides a good way of application adaptability, offering a way of service discovery and dynamic customisation.

3.12 Spring AOP⁹

Spring [99][100] is a layered Java/J2EE application framework. A central focus of Spring is to allow for reusable business and data access objects that are not tied to

⁹ Web Page, <http://www.springframework.org/>

specific J2EE services. Such objects can be reused across J2EE environments (web or EJB), standalone applications, test environments, etc. without any hassle. We focus on AOP functionality provided by Spring. Spring is currently being developed as Open Source project.

3.12.1 Programming Model

Spring is a framework, so AOP features are managed implementing interfaces. Spring AOP is based on *pointcut interfaces*, *advice interfaces* and *advisors interfaces*, which encapsulates both pointcut and advice. Advisors are managed through AOP proxies.

Primary entities in Spring are beans in Spring sense, that are POJO's with some properties. AOP can be applied to beans out or inside a IoC (Inversion of Control) container.

Spring is implemented in pure Java, so there is no need for special compilation process. Spring AOP doesn't need any control over the class loader, and is suitable in a J2EE web container or application server.

Joinpoint Model and Configuration of Pointcuts

Currently, Spring only supports interception of method invocations. Interception of field access can be easily added, but Spring authors consider that it violates OO encapsulation, being not wise in application development.

Pointcuts are declared by implementing interfaces provided by the framework, where methods to match method invocation must be implemented. Union and intersection of pointcuts is available. Pointcuts can be either static, based just in class and method name, or dynamic, taking into account method arguments. Static pointcuts can be declared in XML configuration files using regular expressions in a Perl syntax. Metadata attributes can be used in beans, and these metadata can be used in pointcut declaration. An abstract pointcut superclass is provided to allow users define their own pointcuts.

Configuration of pointcuts can be done with *advisors* or in external XML files. An advisor is a modularisation of an aspect, and typically incorporates both an advice and a pointcut. An advisor is constructed implementing an interface. Advisors are managed through AOP proxies, and many configuration parameters can be declared in external XML file. Reusability of advice and pointcuts is achieved because they are implemented in separate classes and join together in another class called advisor. However, configuration of pointcuts can be done most of time just modifying XML files.

Aspect Advice in Spring

In Spring advice are coded by implementing interfaces. There are several kind of advice: *around*, *before*, *throws*, *after returning* and *introduction*. Around advices are the most powerful advices performing actions before and after method invocation. Only one method named *invoke(Invocation invocation)* will encapsulate the aspect

advice. A *proceed()* method is used to proceed down the interceptor chain towards the joinpoint. Using the *invocation* parameter the aspect has access to contextual information about the intercepted joinpoint and is used to drive the advice chain.

Before advice could be implemented through around advice, but in before advice there is no need to invoke *proceed()* method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

Throws advice in invoked after the return of the joinpoint if the joinpoint threw an exception. Spring offers type throw advices.

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

Mixins and other information can be added to beans using introduction advices. They are only applied at class level, rather than a method level.

Aspect Instantiation Mode

Aspect in Spring can be created *per class* or *per instance*. In a per class aspect, advice are shared by all advised objects. In a per instance aspect there is a unique instance of the aspect for each advised object.

Aspects are managed through proxies. A proxy can contain any number of beans and advisors. If you're using the Spring IoC container, you must use a Spring's Factory Bean, so advices and pointcuts can also be managed by IoC.

3.12.2 Platform Infrastructure and Services

Spring can be integrated with any application server. Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a J2EE web container or application server.

Infrastructure of Spring

As a framework Spring does not provide any infrastructure, relying in the J2EE application server you choose to run Spring.

Spring adds a higher layer of abstraction that is the "Bean Factory." A Spring bean factory is a generic factory that enables objects to be retrieved by name, and which can manage relationships between objects. Each bean definition can be a POJO (defined by class name and JavaBean initialisation properties), or a FactoryBean. The FactoryBean interface adds a level of indirection and is used to create AOP proxied objects.

One of the most important features of this framework is the IoC (Inversion of Control) container, a lightweight container. IoC moves the responsibility for making things happen into the framework, and away from application code. Where configuration is concerned this means that while in traditional container architectures such as EJB, a

component might call the container to say "where's object X, which I need to do my work", with IoC the container figures out that the component needs an X object, and provides it to it at runtime. The container does this figuring out based on method signatures (such as JavaBean properties) and, possibly, configuration data such as XML.

IoC has several important benefits:

- Because components do not need to look up collaborators at runtime, they're much simpler to write and maintain, and to test.
- A good IoC implementation preserves strong typing. If you need to use a generic factory to look up collaborators, you have to cast the results to the desired type. With IoC you express strongly typed dependencies in your code and the framework is responsible for type casts. This means that type mismatches will be raised as errors when the framework configures the application.
- Most business objects don't depend on IoC container APIs. This makes it easy to use legacy code, and easy to use objects either inside or outside the IoC container.

Services of Spring

Spring includes:

- Generic abstraction layer for transaction management, allowing for pluggable transaction managers. Spring's transaction support is not tied to J2EE environments.
- JDBC abstraction layer that offers a meaningful exception hierarchy.
- Integration with Hibernate [144], JDO [145], and iBATIS [146] SQL Maps.
- AOP functionality, fully integrated into Spring configuration management. You can AOP-enable any object managed by Spring.
- Flexible MVC (Model-View-Controller) web application framework, built on core Spring functionality.

3.12.3 Current Status of Development

Spring framework is currently released under version 1.1 as free software. Version 1.2 is under development. Spring, as a framework, have many future works in progress, but we are interested in AOP future additions overall. In this sense, Spring main efforts are focused in performance improvements, taken into special account proxy optimisation, and providing more expressive pointcuts.

Some tools are being developed as subprojects of Spring, like an IDE for Eclipse and a BeanDoc, and the called Rich Client platform. The goal of the spring-rich client project is to provide a viable option for developers that need a platform and a 'best-practices' guide for constructing professional Swing applications quickly.

Some features of the Rich Client platform under development include:

- An action framework that provides centralised configuration of Swing actions and appropriate handler registration based on the current active view.

- Support for multiple window management, page configuration, and view management.
- Common support classes addressing various rich client requirements including: well formed dialogs, wizards, input validation (typing hints and validation results reporting), button bars, internationalisation, image/icon caching, progress monitoring, UIthreading (classes cleanly promoting responsive UIs), treetable/property sheet, table sorting/high-volume table updates, GUIstandards builders/helpers, help/about, etc.

3.12.4 Evaluation

Spring is a J2EE lightweight framework that adds some interesting features to common J2EE application servers. It relies on J2EE technology, so the evaluation discussed in section 2.5 regarding J2EE middleware is still valid here.

Customisability is like J2EE middleware; therefore extension with respect binding types is unsupported in Spring, although a broad collection of binding types are provided. A simple binding adaptation can be achieved using JMS facility of J2EE. No special resource management and policy extension is added in relationship with J2EE. Flexibility is quite high, being that Spring is an extensible framework. AOP proxies can be configured at runtime, adding or withdrawing advisors, so application behaviour can be modified at runtime if the required advisors are written.

Binding management is the key improvement offered by the Spring framework, whereby through the IoC container, EJB management and collaboration is simplified. The improvement in ease of use with respect to traditional J2EE is another point of interest for Spring developers.

In relationship with AOSD criteria, Spring aspects can be applied over simple beans, by writing an implementation of interfaces. The Spring joint point model is open and extensible, although the authors recommend intercepting public messages from a beans interface. Pointcuts and advice are coded independently and later joined together in advisors. Most of pointcut declarations can be done via external XML files. Being separated entities advices and pointcuts, a high degree of reusability can be achieved. AOP is managed by AOP framework through proxies, in a dynamic, non-invasive weaving mechanism. Advisors can be added and removed from proxies at runtime.

3.13 WSML¹⁰/JAsCo¹¹

The WSML (Web Services Management Layer) [101][102][103] is a middleware layer tailored for the integration, selection and client-side management of web services and service compositions in applications. The WSML is built around the dynamic AOP language JAsCo [104][105]. WSML and JAsCo have been developed by the System and Software Engineering Lab at the Vrije Universiteit Brussel.

¹⁰ Web page, <http://ssel.vub.ac.be/wsml>

¹¹ Web page, <http://ssel.vub.ac.be/jasco>

3.13.1 Programming Model

The WSML focuses on the integration of web services in client applications, without hardwiring the services and their interfaces in the client. All service-related code is taken out of the client and placed in the WSML. Using aspects, a flexible redirection mechanism has been developed allowing to dynamically integrating web services and service compositions. The mechanism allows hot swapping between semantically equivalent services whenever a service becomes unavailable, or whenever a service that better fits the criteria of the client becomes available. A selection mechanism makes it possible to specify selection policies based on non-functional properties and behavioural properties of services in order to guide the process of selecting the most optimal service for a given request. Furthermore, support is offered for a wide range of service management concerns that need to be dealt with at the client side. Examples include monitoring, logging, caching, billing, fault tolerance, security, pre-fetching, load balancing, etc. Each of the selection policies and management concerns are encapsulated by aspects.

The WSML relies on JAsCo for the deployment and execution of its aspects. The JAsCo language is an aspect-oriented extension to Java that introduces two new concepts: *aspect beans* and *connectors*. Aspect beans encapsulate crosscutting concerns independently of concrete component types. Connectors deploy one or more aspect beans within a concrete component context and allows specifying an explicit combination of their aspectual behaviour.

Joinpoint Model and Configuration of Pointcuts

The joinpoint model of JAsCo is very similar to AspectJ's, except that JAsCo does not support field level joinpoints (e.g. field set or get) at the moment. A rich pointcut language is available to define aspect applicability in a declarative way. In contrast to AspectJ, JAsCo pointcuts are split in an abstract pointcut description in terms of *abstract method parameters* that are bound to one or more concrete methods when the aspect is deployed in a connector. The connector language supports a wide range of facilities to declaratively specify a set of methods like wildcards, subtype matching, limited subtype matching (only methods defined in the supertype), detailed matching on Java 1.5 annotation types and values etc...

The JAsCo pointcut language also supports stateful aspects, whereby aspects are triggered depending on the sequence of events. This is specified using a *deterministic finite automaton* (DFA) like language. Advices can be attached to any transition in the DFA. The main limitation of the stateful aspects language is that the sequence has to be regular. Supporting non-regular languages is work-in-progress.

Aspect Advice in WSML

JAsCo supports the following advices:

- **before()**: executes before joinpoint. Always void return type.
- **after()**: executes after joinpoint regardless of how the joinpoint executes (exception or normal return). Always void return type.

- **around()**: executes around joinpoint. Return type is Object.
- **after throwing(MyException x)**: Executes after joinpoint when an exception of type MyException is thrown. Always void return type.
- **after returning(String x)**: Executes after joinpoint when an object of type String is returned. Always void return type.
- **around throwing(MyException x)**: Executes around joinpoint when an exception of type MyException is thrown. Allows to alter thrown exception, rethrow it or return normally. Return type is Object.
- **around returning(String s)**: Executes around joinpoint when an object of type String is returned. Allows to alter return value. Return type is Object.

The difference between after returning/throwing and around returning/throwing is that the around counterparts allow altering the returned value and the thrown exception (or not throw an exception at all) while the after advices can not affect the returned value and thrown exception.

When a stateful pointcut is used, advices can be attached to any transition in the stateful pointcut. An advice that is attached to a specific transition will only execute when that transition is fired. As such, multiple advices of the same type might be defined for different transitions. It is of course also possible to specify a global advice for all transitions.

The JAsCo advice language is basically the Java language extended with a limited set of keywords:

- **thisJoinPoint** for reflection about the current joinpoint
- **thisJoinPointObject** allows accessing the object belonging to the current joinpoint (i.e. this with call and target with execution).
- **proceed** in around * advice for invoking the next advice or original behaviour

Aspect Instantiation Mode

JAsCo aspects are always explicitly instantiated in the connector. This allows for detailed aspect instantiation and initialisation control. As such, it is possible to instantiate the same aspect multiple times in, for example, different contexts or with different properties.

In case automatic creation of multiple aspects is required, the following keywords can be used for annotating the instantiation expression in the connector:

- **perobject**: one unique hook for every target object instance
- **perclass**: one unique hook for every target class
- **permethod**: one unique hook for every target method
- **perall**: one unique hook for every joinpoint
- **perthread**: one unique hook per executing thread
- **per(CustomFactoryClassName)**: A custom factory implementation implementing the IAspectFactory interface.

3.13.2 Platform Infrastructure and Services

The WSML relies on the JAsCo runtime weaver for the dynamic deployment of aspects. JAsCo employs a genuine run-time weaver that weaves, unweaves and reweaves aspects entirely at run-time. The JAsCo run-time weaver employs a custom byte-code replacement framework based on JDI or JPLIS depending on the VM's capabilities. Aspects can be flexibly added/removed during run-time of an application, even remotely. The run-time performance of the run-time weaver is able to compete with AspectJ and AspectWerkz and in some cases it is able to improve considerably.

JAsCo is integrated in the AWBench project of the AspectWerkz team. These benchmark results indicate that JAsCo is always at least as fast as AspectJ and AspectWerkz. Around advices are executed about 15 times faster than AspectWerkz due to aggressive around advice optimisations like advice inlining. When combining several aspects at the same joinpoint, the run-time performance of JAsCo is about 5 times faster than AspectJ and about 10 times faster than AspectWerkz. The predefined aspect factories (per*) are also heavily optimised. For example, perobject executes six times faster than the counterpart in AspectWerkz. The weave-time of the run-time weaver is, in case no around advices are present, similar to AspectWerkz's weave time. When around advices are present, the weave-time per joinpoint is about 50% slower than AspectWerkz due to the aggressive around advice optimisations.

More information and benchmarks on the JAsCo weaver:
<http://ssel.vub.ac.be/jasco/documentation:ruw>

Infrastructure of the WSML

The architecture of the WSML is displayed in figure 16. Core repositories of the WSML keep track of the deployed aspects and manage their state. If necessary, required aspect code and connector code is generated automatically. A pluggable server layer allows connecting with several application servers. Administration of the WSML is possible through a set of management tools ranging from a web based interface, xml-based configuration files to remote web service invocations.

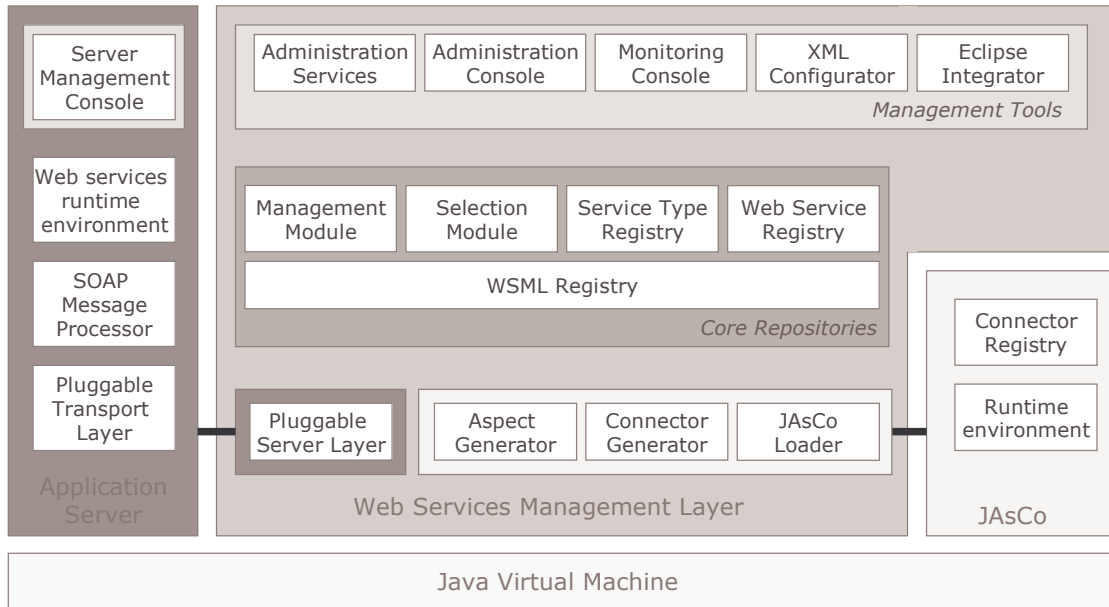


Figure 16. Architecture of the Web Services Management Layer

Services of the WSML

From the point of view of the software developer, the WSML offers a wide range of reusable services, such as service selection policies, service caching, logging and monitoring. These crosscutting management concerns can be easily reused in different contexts as they are modularised in reusable JAsCo aspects. The WSML has tool support for manual and automatic generation of JAsCo aspects and connectors.

3.13.3 Current Status of Development

Full working prototypes of JAsCo and the WSML are available. JAsCo is still work-in-progress, albeit already very functional and stable. JAsCo consists of the following tools:

- Command-line Compilers
- Run-time Infrastructure (including the weavers)
- Eclipse Plug-in (status: beta)
- Eclipse CME Plug-in (status: proof of concept)

The current version of the WSML offers support for the dynamic integration of web services and service compositions through the use of redirection aspects. A reusable library for service selection policies and client-side management concerns is also available. The WSML can be deployed in two scenarios. Firstly, the WSML can run in the same virtual machine as an additional internal component. Secondly, the WSML can run as an independent server in a large scale environment where multiple clients need to be managed. The WSML also offers a plug-in for the integration with the Eclipse IDE.

3.13.4 Evaluation

In this approach, the *flexibility* of the client application is improved as web services and service compositions are no longer hard-wired. At runtime, new services and compositions can be specified and integrated in the client. Also, additional selection-policies and management concerns that were not anticipated at design or deployment time can be integrated non-invasively.

Regarding *performance*, triggering at least one aspect for each web method invocation induces some overhead in comparison with a direct invocation to a static stub. However, as the current state of the JAsCo technology requires little overhead to trigger an aspect, in comparison to a regular method invocation, the expected overhead of employing the WSML is not very significant in relationship to the overhead generated by the network or the performance cost for marshalling and unmarshalling SOAP messages. Our approach is also more optimal than other dynamic approaches such as Dynamic Proxies and Dynamic Proxy Invocation as these require generating a new stub every time a hot-swap between services takes place. Our approach is pro-active as an aspect and stub are generated and compiled before they are added to the redirection mechanism. Also, additional selection policies can be introduced to pro-actively select the most optimal service for a given request.

With respect to *reliability*, the WSML introduces a range of fault tolerance aspects that are triggered in specific scenarios (e.g. services do not respond, network becomes unavailable). These aspects can deal with all kinds of service related failures and recover from them by triggering a range of actions ranging from hot swapping to another service, returning a cached result or raising an alert. This enhances the robustness of the client application as the client does not depend on the availability and results of a single service anymore.

With respect to the *criteria used to compare middleware platforms for AOSD*, WSML is a non-invasive approach developed to apply aspects to web services. WSML relies on the JAsCo language which applies the separation of concerns to both object-oriented applications and JavaBeans components. The weaving mechanism offered by JAsCo is dynamic, weaving components and aspects at runtime. JAsCo separates the definition of pointcuts from the definition of advice promoting the reusability of aspects in different contexts. Since the weaving of aspects is performed at runtime, final applications are highly adaptable in JAsCo.

3.14 Evaluation Overview

This section presents first an overview of the evaluation of the different middleware platforms for AOSD according to the criteria of flexibility, reliability and performance used to compare middleware platforms in section 1 of the document. Secondly, it presents an overview of the evaluation of the same middleware platforms according to the specific criteria of middleware for AOSD that were described in section 3.2.

For each evaluation, the section presents first a quantitative overview of the evaluation and then a more detailed platform comparison and general discussion

3.14.1 Evaluation According to General Criteria

3.14.1.1 Quantitative view

Table 2 provides a quantitative view of the preceding evaluation of platforms according to the general criteria of flexibility, reliability and performance.

The rest of criteria, included in Table 1, have been discussed in the evaluation of each platform if they can be applied. The guidelines followed to classify each platform in section 2.5 are also followed here.

TABLE 2. Middleware for AOSD platform comparison with respect to general criteria

	Flexibility	Reliability	Performance
AO4BPEL	●	●	●
AspectJ2EE	●	●	●
AspectWerkz	▲	●	▲
CAM/DAOP	●	▲	●
JBoss AOP	▲	▲	●
JAC	●	○	●
Lasagne	●	▲	○
PRISMA	●	●	—
PROSE/MIDAS	●	○	●
Spring AOP	▲	—	●
WSML/JAsCo	●	▲	▲

▲	very high
●	sufficient
○	limited
▽	very low
—	Information is not available

3.14.1.2 Discussion

With respect to the definition of *flexibility* provided before, most of the platforms do not satisfy the three constraints of static and dynamic customisation of the platform

environment, the support for extensions and support for large-scale customisations. There is however a considerable increasing of flexibility in AO platforms regarding the middleware platforms described in previous section. First of all, all AO platforms in this survey solves the limitation that traditional middleware platforms have when they offer only a fixed set of common services. This is highly important from the point of view of application developers, which can develop their applications focusing only in the application domain, and making use of the common services offered by the platform. This common services are offered in some platforms as library of aspects, that can be used at the application level (e.g. CAM/DAOP, JAC, PROSE, etc), while in other platforms these services can also be incorporated at the middleware level as extensions of the functionality of the middleware (e.g. JBoss AOP). In addition, these platforms offer the appropriate mechanisms to easily extend the number of aspects provided by the platform developer/vendor. This means that application developers can incorporate new aspects as they are needed.

Among the different platforms, JBoss AOP and Spring are the most flexible platforms evaluated. The JBoss AS supports extension and large-scale customisation based on the use of the JMX microkernel, where all the functionality of the platform is provided homogeneously as plug-ins. Services can be added or removed from the microkernel as needed. Spring can also be extended and supports large-scale customisations by the use of the IoC container. Other platforms based also on the EJB/J2EE model, such as AspectJ2EE, AspectWerkz and Spring, improve the low flexibility offered by the standard J2EE platform by separating in aspects common middleware services. The main difference with JBoss AS is that they do not define a homogeneous model that allows separating using the same mechanism both the common services usually provided by the application server container and the new services separated with AOP.

Finally, as mentioned before if we compare the middleware platforms in the first part of the survey with the AO platforms in this section, the later are in general more flexible than the prior. In fact, several of the AO platforms (AspectJ2EE, JBoss AOP, AspectWerkz) are the result of trying to cope with the limited flexibility of J2EE.

Reliability is especially important in AO platforms since most of them allow to add and remove aspects at runtime. The consequence of such degree of adaptability may be the introduction of possible inconsistencies during the execution of the application. Therefore, with respect to *reliability*, an AO platform is considered to be highly reliable if it manages possible inconsistencies when an aspect (or other kind of services in the platform) is added or removed from an application at runtime. This consistency management can be provided by the platform itself, through specific mechanisms developed for this purpose, as is the case of CAM/DAOP and Lasagne, or it can be provided by defining specific *reliability* and *fault tolerance* aspects as is the case of other approaches such as AO4BPEL and WSML/JAsCo. Other approaches such as AspectJ2EE does not need to check the consistency at runtime since aspects can not be modified after the deployment of the application.

There are some approaches such as JAC and PROSE/MIDAS that do not provide explicit support for consistency management at runtime. However, in order to solve the problem of inter-aspect composition, JAC provides some level of consistency by allowing the use of a *composition aspect*, which can be used to define relationships

among aspects and to perform checks before aspects are executed. As mentioned before, there are AO platforms that use the underlying infrastructure offered by a commercial common platform. This is the case of AspectJ2EE, AspectWerkz and Spring AOP, which rely on the infrastructure offered by J2EE and JBoss AOP, which is J2EE compliant. These platforms should leverage the reliability offered by J2EE with new mechanisms to manage consistency when adding/removing aspects.

With respect to *performance*, we focus on the overhead that may be introduced by the weaving mechanism offered by the platform. In this sense, most of the middleware platforms use a dynamic weaving mechanism and dynamic weaving usually introduces some level of overhead in the system. Therefore, comparing AO platforms with generic middleware platforms with respect to performance, the dynamic weaving mechanism is one of the most important factors decreasing the performance of AO platforms. However, most of the platforms care about this issue and part of their research is focused on reducing this overload. Examples of these platforms are AspectWerkz, JBoss AOP, PROSE/MIDAS and JAsCo which are releasing new versions in which their original weaving mechanisms is adapted to improve performance.

The AspectWerkz team has recently released the AWBench benchmarking to performance test a variety of different AOP implementations. Some of the middleware platforms in this survey have been part of this comparison and can be found in <http://docs.codehaus.org/display/AW/AOP+Benchmark>. Also in [132] a comparison of different AO platforms can be found, including JAsCo, AspectWerkz, JBoss and Spring AOP. This comparison also includes the benchmark for a non-advised application, being able to notice that the overhead of AOP approaches exist but, in some cases, it is not excessively relevant for the performance of the final application.

In the case of AspectJ2EE, performance is not affected by dynamic composition since the weaving is performed at deployment time and, therefore, no overhead is introduced at runtime. There are also approaches in which the overhead is minimal in comparison with the overhead generated by the communication and transport mechanisms they require. This is the case of AO4BPEL and WSML/JAsco specially implemented to support the development of aspect-oriented web services. This is also the case of CAM/DAOP which is mainly used to develop complex web-based applications.

Finally, Lasagne is the only platform that reports low performance. However, an important advantage of Lasagne that may introduce some extra overhead, in addition to the overhead introduced by dynamic composition, is the very high reliability offered by the platform.

3.14.2 Evaluation According to AO Specific Criteria

3.14.2.1 Quantitative view

According to the criteria described in section 3.2, table 3 summarises the characteristics of each middleware platform that was studied in this section regarding its programming model, the primary entities supported, the weaving model, the joinpoint model, the aspect reusability and the final application adaptability.

TABLE 3. Middleware for AOSD platform comparison with respect to AOSD criteria

	Programming Model	Primary Entities	Weaving Model	Joinpoint Model	Aspect Reusability	Application Adaptability
AO4BPEL	E	WS	D	NI	—	▲
AspectJ2EE	E	C(EJB)	D	I	—	—
AspectWerkz	E ¹²	O	C, L, R	I	—	▲
CAM/DAOP	N	C	C, R	NI	▲	▲
JBoss AOP	E ¹³	O, C(EJB)	C, L ¹⁴	I	—	▲
JAC	E	O	R	I	▲	▲
Lasagne	N	C	R	NI	—	▲
PRISMA	N	C	R	NI	▲	▲
PROSE	N	O	L,R	I	—	▲
Spring	E ¹²	O ¹⁵	R	NI	▲	▲
WSML/JAsCo	N	WS/O,C(JavaBeans)	R	NI	▲	▲

Programming Model	N – New E – Extension of/Based on standard model	
Primary Entities	O – Objects C – Components C(model) – Component Standard Model	WS – Web Services A – Agents Oth – Other kind
Weaving Model	C – Compile Time D – Deploy Time L – Load Time R – RunTime	
Joint Point Model	I – Invasive NI – Non invasive	
Aspect Reusability	▲ – High (Always) — – Medium (Depends on the software developer) ▼ – Low (Never)	
Application Extensibility/Adaptability	▲ – High (Both runtime and previous development phases) — – Medium (Only one of the alternatives) ▼ – Low (Neither of them)	

As summarised in the table legend, an “N” in the *programming model* column indicates that the middleware platform defines the programming model from scratch, while an “E” indicates that it is an extension or that it is based on an existing standard object/component model.

¹² Relies on the infrastructure offered by a J2EE application server

¹³ JBoss is a J2EE compliant application server

¹⁴ Though weaving is performed at load time by byte-code transformation, there is a Dynamic AOP API that allows adding/removing advices and interceptors on any joinpoint that was aspectised during compilation or load time.

¹⁵ Objects with some special restrictions, such as the definition of their interfaces

In the *primary entity* column an “O” is used to indicate that aspects are applied to objects, a “C” to indicate that aspects are applied to components, a “C(model)” to indicate that aspects are applied to components in a particular component model (e.g. EJB, CCM, etc.), an “A” to indicate that aspects are applied to agents, a “WS” to indicate that aspects are applied to web services and finally, “Oth” to indicate that aspects are applied to any other kind of software artefact.

In the *weaving model* column the weaving can be at compile-time (“C”), at load-time (“L”), at deploy-time (“D”) or at runtime (“R”). More than one are possible. The *joinpoint model* column will indicate if the joinpoint model is invasive (“I”) or non-invasive (“NI”).

In the *aspect reusability* column, aspects are highly reused (▲) if aspects are always reusable independently of the mechanisms that software developers use to develop them. The aspect reusability is labelled as medium (—) if the middleware platform offers mechanisms to both develop reusable and not reusable aspects and it depends on the software developer to build their aspects as reusable as possible. Finally, if a particular approach does not provide any mechanism at all to build reusable aspects, the aspect reusability will be low (▼).

Finally, in the *application adaptability* column we indicate if applications are highly adaptable (▲), medium adaptable (—) or offer low adaptability (▼). To be highly adaptable, the middleware platform has to provide mechanisms to adapt the behaviour of applications in all the phases of the software lifecycle, including runtime. If adaptability is provided only in the early phases of the development, the capacity to adapt an application in that approach will be medium. If no adaptability at all is provided the capacity to adopt an application is low.

3.14.2.2 Discussion

This section compares the different middleware platforms for AOSD and discusses the main similarities and differences among them.

With respect to the *programming model* they define, the number of proposals that are based on a standard object/component model and the number of proposal that define a new model from scratch is similar. Among the prior, all of the studied middleware are based on the EJB component model, though only JBoss/AOP defines a J2EE compliant application server. Two of these, AspectWerkz and Spring, rely on the direct use of a standard J2EE application server, although they provide limited additional support as a middleware layer. Spring introduces special support for middleware, introducing the concept of IoC container, and the new version of AspectWerkz introduces additional middleware support through the concept of the Aspect Container. They have been included in the survey since they have been specially developed to enhance the behaviour of J2EE middleware platforms providing support to separate any kind of common services as aspects.

With respect to the *primary entities* considered by the different platforms, most of them are focused on applying the separation of concerns to objects or components. Only two of them, AO4BPEL and WSML/JAsCo support the development of aspect-

oriented web services. Those platforms that define their own component models such as CAM/DAOP, Lasagne and PRISMA follow the standards of component-based models (e.g. CCM) and defines their components in terms of the interfaces that describe their behaviour (their provided interfaces) and also their dependencies of the environment (their required interfaces). Additionally, most platforms consider that aspects are an additional dimension that is weaved with a core dimension (objects, components, web services). This core dimension encapsulates the core functionality of the application. Only PRISMA provides a different approach, where functionality is considered another aspect and a component is composed by a set of aspects, including the functional aspect.

With respect to the *weaving model* all the platforms offers mechanisms to support some degree of adaptability after the compilation phase. Though three of them, AspectWerkz, CAM/DAOP, and JBoss AOP offer a compilation-time mechanism, they also offer alternatives to weave aspects at load-time (JBoss AOP and AspectWerkz) or at run-time (AspectWerkz and CAM/DAOP). Furthermore, additionally to more traditional weaving mechanisms at compilation-time, load-time and run-time, AO4BPEL, AspectJ2EE and PROSE supports the weaving of aspects during the deployment of the application.

With respect to the *joinpoint model* the important issue is not just if they define a non-invasive or an invasive model. The most important issue is if the joinpoint model is the most appropriate regarding the primary entities affected by aspects. In this sense, most of the approaches consider highly important to avoid violating the object-oriented and the component-based encapsulation. This is especially relevant in the case of components, which are always described by a set of interfaces and that should only be affected by aspects at those joinpoints that are visible through these interfaces. In this sense, all the middleware platforms that define a component and based approach also defines a non-invasive joinpoint model avoiding the interception of internal and private states of components. Only JBoss/AOP and AspectJ2EE define an invasive joinpoint model even when its component model is EJB.

With respect to *aspect reusability* most of the studied middleware platforms provides some kind of support to reuse aspects in different contexts. This is basically achieved by separating the declaration of pointcuts and the definition of advice in two different entities. Also the use of a declarative language, for instance XML, is broadly extended for the declaration of pointcuts. According to this criterion of separating pointcuts and advice, aspects defined in CAM/DAOP, JAC, PRISMA and Spring are always reusable in different contexts. In other proposals such as AspectJ2EE, AspectWerkz, JBoss AOP and Lasagne the reusability of aspects is also promoted. The difference is that these approaches offer both possibilities, relying on the software developer the decision to implement aspects with the goal of being reused in different contexts or not. A different approach is followed by AO4BPEL and PROSE where both pointcuts and advice are defined in the entities. This entity is an XML document in the case of AO4BPEL and Java classes in the case of PROSE.

With respect to *application adaptability* most middleware platform provides support to modify the number of aspects that are applied to an application at load-time or at run-time. Only AspectJ2EE does not allow adapting the binding between aspects and components once the application has been deployed. However, AspectJ2EE tries to

cope with this limitation allowing the definition of multiple advised versions of the same bean, and choosing which to use at runtime.

3.15 Summary

This section has presented an overview of middleware platforms that offer an aspect-oriented programming model and their evaluation with respect to the requirements identified. The main outcome of the evaluation is that all these platforms increase flexibility through the use of aspects. Concretely, all of them improve the separation of common services normally used by distributed applications and offered by standard middleware platforms such as CORBA and J2EE, enabling applications to use an open-list of services by means of aspects. For most platforms this flexibility is unsatisfactory in areas of binding management, and large-scale customisation. Performance is sometimes affected by the weaving mechanisms, though it can be compensated by the high degree of flexibility and adaptability obtained in final applications.

4. Using AOSD to Structure Middleware

In contrast to section 3, which surveyed middleware supporting AOSD, this section reviews research into using AOSD for the construction of middleware. Rather than presenting the use of AOSD from the point of the application developer it is considered from the middleware designer's perspective. The survey includes the use of AOSD within a single middleware layer as well as provision of services to the layer above.

In evaluating the contributions from the research in this survey the categories of flexibility, reliability and performance are used. The main purpose of AOSD is flexible and reliable composition with acceptable impact on performance. When considering flexibility we note when AOSD has been used to address problems peculiar to middleware construction. We also note when AOSD has been used in the provision of a reliable platform for applications. Finally many complex performance features can only be implemented reliably if AOSD is used, especially if they are to be applied dynamically.

The survey is broken down into four constituent parts reflecting the inherently layered structure of middleware:

- a. The use of aspect-oriented software development techniques to construct host-infrastructure middleware. Examples of host-infrastructure include virtual machines and operating system software.
- b. The use of aspect-oriented software development techniques to construct distribution middleware, for example, Object Request Brokers and messaging systems.
- c. The use of aspect-oriented software development techniques to construct middleware services. These include transaction and persistence services and security.
- d. The use of aspect-oriented software development techniques to construct domain-specific middleware services.

The following sections detail the work surveyed in each category.

4.1 Host-Infrastructure middleware

Although operating systems software itself is not considered middleware, research into using AOSD techniques in its construction are included where it relates to services provided to the layer above or where the techniques are equally applicable to higher layers. The performance of the lowest middleware layer is critical because of its impact on those above. We are equally concerned with the ability to implement performance features using AOSD as we are with the impact of using the technique to facilitate flexible or reliable composition.

Extensive research has been carried out using the FreeBSD operating system [106][107][108]. The primary goal of using AOSD is the flexible composition of OS features. The tension between performance and reliability is explored by using an

aspect to implement path-specific optimisations such as page pre-fetching. This approach addresses the frequent need to violate layered architectures when using dynamic context information.

The Organic Aspects for System Infrastructure Software (OASIS) [109] project looks at modularising evolving concerns while keeping an eye on performance. The garbage collection in the Jikes VM is used as an example. In particular AOP is used to implement different collection algorithms.

The Linux kernel is used as an example of applying AOSD to address the issue of supporting product variants [110]. While the framework used to support existing programming approaches is good, the patches required to implement it are not. This work is an example of using AOSD to resolve the frequent tension between flexibility and reliability in middleware systems while highlighting the importance of acceptance especially with respect to both runtime and compile-time performance. The research makes a distinction between static and dynamic AOP mechanisms and their possible optimisations.

The Aspects in Operating Systems: Tools and Applications (AOSTA) Project investigates the use of AOP to address operating system performance and reliability. In particular it focuses on the implementation of autonomic computing which is inherently crosscutting and where the long term maintenance and reliability of traditional programming approaches are at odds with the goals of organic computing. The TOSKANA [111] toolkit allows the deployment of dynamic aspects for kernel functions, in this case NetBSD. Further work [112] seeks to address the problem of a restricted joinpoint model in native code. The use of a low level virtual machine provides dynamic AOP functionality and aspect activation. This is made possible through use of microkernel architecture which mediates all interaction with hardware.

Providing autonomic-like capabilities using AOP are also explored in an extension to Arachne [113] a dynamic weaver for C. In particular the ability to address middleware reliability issues such as security flaws by dynamically deploying aspects are described. Certain shortcomings in the expressiveness of existing pointcut languages are also examined.

The evolution of a large piece of software is particularly complex even when the necessary customisations appear to be systematic. Adding support for the Bossa [114] processor scheduling framework to the Linux kernel is used to illustrate the use of AOSD. A rules based approach is used to identify both where and when events should be generated.

To facilitate the extension of a legacy system it can be viewed as a single component in a component model supporting a well defined extension mechanism. The specification of the extension interface, a rules based approach to identifying where it is invoked in the legacy system and tools for automating the necessary code customisations, demonstrates the use of AOSD to solve this problem [115]. The examples of adding new scheduling policies to Linux and pre-fetching to the Squid Web cache are used.

4.2 Distribution Middleware

The purpose of this middleware layer is to provide transparent distribution of application function. The two examples of such a middleware system are ORBs and messaging systems which address distribution at different levels of granularity. Both have concerns including the need to support different transport protocols, conversion of data types and performance issues such as caching.

The problem of feature growth and code bloat is inherent to most software including middleware. The experience of using AspectJ [116] and AspectIDL [117] for footprint and feature management [116] in order to address the knock-on effects to maintainability and performance is reported. Subsets of an event channel based on a simple Java implementation are used as an example.

The task of customising a database to the needs of a particular application or organisation has been explored using aspects [118]. The research shows how the required changes to implement customisable features are localised and can be made at either compile or run-time.

The implementation of non-functional crosscutting concerns and in particular the implications for heterogeneous systems are explored with DADO (Distributed Adapters for Distributed Objects) [119]. Adapters are specified in an IDL allowing them to be implemented in a language compatible manner with the target application. The particular examples of security and caching used in this work have location-specific requirements and their implementations as aspects and deployment at runtime demonstrates the flexibility achieved through AOSD.

Middleware vendors create generalised platforms to satisfy the needs of a wide variety of applications. Several open source ORB implementations [2][120][121][122][123] are the target of a refactoring exercise to assess the benefits of AOP for modularity and its impact on system performance. Middleware-specific concerns such as dynamic invocation and supporting portable interceptors as well as more common aspects such as error handling are extracted using aspect mining techniques. The tension between performance and reliability is explored through objective measurements of structural complexity.

Most existing middleware implementations achieve a separation of concerns between business logic and the technical infrastructure necessary to run it through the use of container services. However there is a heavy reliance on an often complex programming model and little scope to extend the set of services provided to include arbitrary application level aspects. Furthermore the middleware services themselves are generally not well modularised making it difficult to adapt the platform to application specific needs or swap the implementation of a particular service. Alice [124] proposes a minimal container approach using aspects and Java 5 annotations to provide crosscutting services with business logic written using plain Java interfaces and classes. The result is a more flexible solution allowing composition of new services and reuse of application components.

One barrier to the creation of product lines comprising reusable components from different sources is the provision of platform specific qualities of service such as

problem diagnosis or performance monitoring. “Using AspectJ for Component Integration in Middleware” [125] demonstrates the use of AspectJ to provide tracing, first-failure data capture and performance monitoring services for a component to be released under an open source license. The need to support separate code bases is avoided by implementing proprietary services, which could not be included in the open source version, as aspects.

Even the most well modularised systems suffer from the complex interactions between components and the shortcomings of their existing OO implementations. A successful project to separate the EJB container [126] from the rest of the application server demonstrates the benefits of an iterative aspect-oriented approach. The built-time performance and scalability of tools to support AOP are explored when targeting a project comprising over 15000 source modules.

Applications operating in highly dynamic environments must adapt to various factors including resource availability. Adaptation at runtime can be considered a concern that crosscuts an application and is the ultimate expression of flexibility and reliability. A framework based on the Fractal [127] component model is used to explore adaptive caching policies in an image viewer application.

4.3 Middleware Services

The primary purpose of a middleware platform is the provision of services such as security and persistence to user written applications. In contrast to Part 3 of this report we now survey research into using AOSD to construct these services rather than the manner in which they are presented to the application.

Web caches are critical in reducing user response times for a given network latency and bandwidth. Their performance can be improved through the use of pre-fetching but to achieve the best results policies must be adapted to user characteristics and the Web itself. The use of dynamic aspect weaving [128] can facilitate the level of flexibility required.

Research into implementing persistence as an aspect [129] explores both the degree to which an application can be developed independently of the persistence concern and the reusability of that concern. The interaction between aspects responsible for tracking persisted objects, object-to-relation mapping and object flattening is considered and the trade-off between flexibility and performance of the solution are described.

JBoss AOP (<http://www.jboss.org/products/aop>) is an open source framework tightly integrated with an application server. An extensive aspect library is used to provide services to applications as well as the implementation of the middleware itself.

4.4 Domain-Specific Middleware Services

In this final section we describe research into the construction of domain specific services such as Enterprise Application Integration (EAI) and workflow. We are interested in the provision of services such as security from the lower levels in the

middleware stack to create a more integrated solution. The work surveyed demonstrates the use of AOP to address the consequences of an invasive programming model.

EAI is primarily used to reduce the cost of integrating a new application into a system already comprising a number of different applications possibly from different implementers. EAI systems typically take the form of an information bus providing infrastructure services, process flow and a shared object model. AspectJ has been used to eliminate code tangling [130] inherent in the update logic associated with the observer pattern, the implementation of proxies for infrastructure services and the creation of a shared object model through the use of static crosscutting.

4.5 Summary

This section presents a survey of research undertaken by partners, and those outside the network, into the use of AOSD to structure middleware. The results have been arranged to reflect the inherently layered structure of middleware, allowing the concerns addressed and techniques used to be compared. Of particular importance is the contribution AOSD can make to resolve the tension between flexibility, reliability and performance: key criteria for the evaluation of middleware.

5. Conclusions

In this report we surveyed the current approaches to middleware and found there were numerous shortcomings with regard to our chosen evaluation criteria, namely *flexibility*, *reliability* and *performance*. Commercial middleware solutions, such as CORBA, J2EE et al, offer very low flexibility, providing little or no support for dynamic change and customisation. In contrast a number of the research middleware platforms surveyed offered high degrees of flexibility, but this was always at the expense of performance and reliability. In essence, none of the middleware platforms surveyed completely satisfied our criteria in a *balanced way*. Additionally, large scale customisability remains completely unaddressed by all surveyed middleware platforms.

Next, we surveyed a range of AO middleware approaches and compared them on the same criteria as well as a number of additional ones relating specifically to AO, namely, *aspect programming model*, *primary entities supported*, *weaving model*, *joinpoint model*, *aspect reusability* and *application extensibility/adaptability*.

Regarding our definition of flexibility in section 2, AO middleware offers significant improvements by providing extendable common services as aspects. These aspects can be used at the application level (e.g. CAM/DAOP, JAC, PROSE, etc.) or middleware level (e.g. JBoss AOP). Moreover, large scale customisation is supported by a number of platforms (e.g. JBoss AOP, Spring, etc.), while the inflexibility of the existing EJB/J2EE middleware platform is improved by separating the common services as aspects in AspectJ2EE, AspectWerkz and Spring.

With respect to reliability, all AO middleware platforms, with the exception of JAC and PROSE provide a means for managing inconsistencies either explicitly, using mechanisms that exist in the platform (e.g. CAM/DAOP, Lasagne, etc.), or by exploiting existing functionality of the J2EE platform (e.g. AspectJ2EE, AspectWerkz, etc.).

Performance of all AO middleware platforms, with the notable exception of Lasagne, was sufficient to very high. Dynamically weaved languages typically introduce a performance penalty; therefore in general these have a lower performance when compared to statically weaved languages. In an effort to improve the situation, many of the AO middleware platforms reviewed are currently undergoing changes to their weaving mechanisms. In general though, it is believed that the performance hit was outweighed by the improved flexibility and reliability offered by the platforms. In essence, we believe the main purpose of AOSD is to allow flexible, reliable and highly modularised composition with acceptable performance.

Finally, we reviewed research into using AOSD in the construction of middleware from the middleware designer's perspective. We also reported on various projects that are successfully utilising AOSD to structure middleware.

Overall, from analysis, it appears that AO middleware techniques offer an improvement over traditional middleware by delivering flexibility, reliability and performance in a *more balanced manner*. However, like traditional middleware platforms, there is an argument to be made that there is too diverse a selection of AO middleware platforms. Therefore the natural step is to work towards a *generic AO middleware architecture* prototype in due course.

To conclude, developing large scale distributed systems is highly problematic. Middleware was developed to facilitate the construction of distributed systems, but has become overly complex and difficult to understand. This complexity is at odds with one of its fundamental aims – to make the building of distributed systems easier. The software decomposition techniques offered by current middleware are insufficient for the integration of concerns that crosscut system structure such as persistence, synchronisation and transactional communication and security. Additionally, achieving effective reuse of these concerns is difficult as they tend to become entangled with the system, decreasing evolvability, and, thus, increasing the possibility for *architectural erosion* [131]. Clearly, this situation is untenable for future development and evolution of large distributed systems.

We believe that the future construction of large scale distributed systems lies with AO middleware. AOSD offers great potential for the creation of distributed systems and complements middleware by easing complexity, and facilitating reuse, large scale customisability and evolvability, therefore, decreasing the likelihood of architectural erosion.

6. Acknowledgements

The authors would sincerely like to thank all the contributions, suggestions and reviews we received for this document.

People who contributed to this document were (in no particular order):

Anis Charfi (Darmstadt)

Geoff Coulson (Lancaster)

Andronikos Nedos and Siobhan Clarke (Trinity)

Mercedes Amor (Malaga)

Maja D'Hondt, Johan Brichau, Wim Vanderperren and Bart Verheecke (Brussels)

Eddy Truyen, Frans Sanen and Wouter Joosen (Leuven)

Tal Cohen (Technion)

Lodewijk Bergmans (Twente)

Lionel Seinturier and Mario Südholt (INRIA)

Christa Schwanninger and Egon Wuchner (Siemens)

References

- [1] A. Colyer, G. Blair, and A. Rashid, "Managing Complexity in Middleware," presented at Workshop on Aspects, Components and Patterns for Infrastructure Software, Third International Conference on AOSD, Boston, USA, 2003.
- [2] C. Zhang and H.-A. Jacobsen, "Refactoring Middleware With Aspects," IEEE Transactions on Parallel and Distributed Systems, vol. 14, 2003.
- [3] R. Filman, T. Elrad, S. Clarke, and M. Aksit, "Aspect-Oriented Software Development", Addison Wesley, 2005.
- [4] Sun Microsystems, Java Servlet Technology, <http://java.sun.com/products/servlet/>
- [5] J. Regehr, M. B. Jones and J. A. Stankovic, "Operating System Support for Multimedia: The Programming Model Matters", Microsoft Research Technical Report, MSR-TR-2000-89, Sept. 2000
- [6] Object Management Group, CORBA/IIOP v3.0.3, OMG Document formal/2004-03-01.
- [7] Object Management Group, Real-time CORBA v2.0, OMG Document formal/2003-11-01.
- [8] Object Management Group, Minimum CORBA v1.0, OMG Document formal/2002-08-01.
- [9] Object Management Group, CORBA Component Model v3.0, OMG Document formal/2002-06-65.
- [10] Object Management Group, Extensible Transport Framework v1.0, OMG Document ptc/2004-03-03.
- [11] Object Management Group, Event Service v1.0, OMG Document formal/2000-06-15
- [12] Object Management Group, Audio/Video Streams v1.0, OMG Document formal/2000-01-03.
- [13] Sun Microsystems, Java 2 Platform, Standard Edition (J2SE) v1.5.0, <http://java.sun.com/j2se/index.jsp>
- [14] Sun Microsystems, Java 2 Platform, Enterprise Edition (J2EE) v1.4, <http://java.sun.com/j2ee/index.jsp>
- [15] Sun Microsystems, Enterprise JavaBeans Specification Version 2.1, <http://java.sun.com/products/ejb/>
- [16] Sun Microsystems, Jini Network Technology Specifications Version 2.0, <http://www.sun.com/software/jini/index.xml>
- [17] G. Czajkowski and T. von Eicken "JRes: A resource accounting interface for Java". Proc. of ACM OOPSLA 98, Vancouver, British Columbia, Oct. 1998, pages 21-35.
- [18] W. Binder, J. Hulaas, A. Villazón and R. Vidal, "Portable Resource Control in Java: The J-SEAL2 Approach", ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001), Tampa Bay, Florida, USA, October 14-18, 2001, pp 139-155.
- [19] F. Guidec and N. Le Sommer, "Towards Resource Consumption Accounting and Control in Java: A Practical Experience". ECOOP'2002, Workshop on Resource Management for Safe Language, June 10-14, 2002.
- [20] M. Jordan, G. Czajkowski, K. Kouklinski and G. Skinner, "Extending a J2EE Server with Dynamic and Flexible Resource Management". Proc. IFIP/ACM/USENIX International Middleware Conference, LNCS volume 3231, Toronto, Canada, Oct. 18-20, 2004.
- [21] K. J. Gough, "Stacking them up: A Comparison of Virtual Machines", Proceedings of the 6th Australasian Conference on Computer Systems Architecture, Queensland, Australia, Jan. 29-30, 2001, pp 55 – 61.
- [22] Microsoft, COM Component Object Model Technologies, <http://www.microsoft.com/com/default.mspx>
- [23] Microsoft, .Net Home Page, Retrieved: Dec. 2004, <http://www.microsoft.com/net>
- [24] E. W. Dijkstra, "Selected Writings on Computing: A Personal Perspective", Springer-Verlag, 1982. ISBN 0387906525.
- [25] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM, vol. 15, pp. 1053-1058, 1972.
- [26] C. Alexander, "The Timeless Way of Building", Oxford University Press (1979), ISBN: 0195024028.
- [27] C. Alexander, "A Pattern Language", Oxford University Press (1977), ISBN: 0195019199.
- [28] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns", Addison-Wesley, 1995, ISBN 0201633612.
- [29] B. C. Smith, "Reflection and Semantics in a Procedural Language", M.I.T. Laboratory for Computer Science Report MIT-TR-272 (1982).
- [30] Smalltalk home page, <http://www.smalltalk.org>

- [31] Python Language home page, <http://www.python.org>
- [32] Ruby home page, <http://www.ruby-lang.org>
- [33] D. McIlroy, "Mass Produced Software Components", Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, pp. 138 - 150.
- [34] G. Heineman and W. Councill, "Component-Based Software Engineering", Addison Wesley, 2001, ISBN: 0201704854.
- [35] C. Szyperski, "Component Software - Beyond Object-Oriented Programming", Addison-Wesley, 1997, ISBN: 0201178885.
- [36] D. Garlan, R. Monroe and D. Wile, "ACME: An Architecture Description Interchange Language", in Proceedings of CASCON'97, November 1997.
- [37] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures", 5th European Software Engineering Conference (ESEC'95), Sitges, Spain, September 1995.
- [38] R. J. Allen, "A Formal Approach to Software Architecture", Technical Report CMU-CS-97-144, May 1997.
- [39] The Stanford Rapide Project homepage, <http://pavg.stanford.edu/rapide>
- [40] R. E. Johnson and B. Foote, "Designing Reusable Classes", Journal of Object-Oriented Programming, June/July 1988.
- [41] M. Fayad and D. Schmidt, "Object-Oriented Application Frameworks", Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.
- [42] M. Weiser, "Hot Topics: Ubiquitous Computing", Computer, Oct. 1993.
- [43] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", Computer, Jan. 2003.
- [44] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "A Taxonomy of Compositional Adaptation," Technical Report MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2004.
- [45] R. Hayton and ANSA Team, FlexiNet Architecture, ANSA Architecture Report, Citrix Systems Ltd., Cambridge, UK, February 1999. Available at: <http://www.ansa.co.uk>
- [46] Ø., Hanssen and F. Eliassen, "A Framework for Policy Bindings", Proc. DOA'99, IEEE Press, Edinburgh, Sept. 5-7, 1999, p 2.
- [47] B. Dumant, F. Dang Tran, F. Horn and J. B. Stefani, "Jonathan: an open distributed processing environment in Java", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer-Verlag, The Lake District, U.K., Sept. 1998, pp 175-190.
- [48] ObjectWeb Consortium, Jonathan v3.0 alpha 10, Oct. 2002, <http://jonathan.objectweb.org/>
- [49] G. S. Blair, G. Coulson, P. Robin and M. Papatomas, "An Architecture for Next Generation Middleware", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK, Springer-Verlag, Sept. 15-18, 1998, pp. 191-206.
- [50] F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G. Blair and G. Coulson, "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications". In Reflection and Software Engineering, Lecture Notes in Computer Science No. 1826, Springer-Verlag, Heidelberg, Germany, June 2000, pp 79-99.
- [51] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas and K. Saikoski, "The Design and Implementation of OpenORB v2", IEEE Distributed Systems Online, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001, http://dsonline.computer.org/0106/features/bla0106_print.htm
- [52] F. Costa and G. S. Blair, "Integrating Meta-Information Management and Reflection in Middleware," Proc. of the Int'l Symposium on Distributed Objects and Applications (DOA'00), IEEE Press, Piscataway, N.J., Sept. 2000, pp. 133-143.
- [53] H. Duran-Limon and G. S. Blair, "The Importance of Resource Management in Engineering Distributed Objects," Proc. 2nd Int'l Workshop on Engineering Distributed Objects (EDO'2000), Springer-Verlag, Berlin, 2000, pp. 44-60.
- [54] K. Saikoski, G. Coulson, and G. S. Blair, "Configurable and Reconfigurable Group Services in a Component Based Middleware Environment", Proc. International SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nürnberg, Germany, October 2000.
- [55] R. M. Moreira, G. S. Blair, E. Carrapatoso, "A Reflective Component-based and Architecture Aware Framework to Manage Architectural Composition", Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01), Rome, September 2001.

- [56] F. Kon, R. Campbell, M. D. Mickunas, K. Nahrstedt and F. J. Ballesteros, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments", 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, August 1-4, 2000.
- [57] F. Kon, T. Yamane, C. Hess, R. Campbell and M. D. Mickunas, "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems", Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas, January, 2001.
- [58] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães and R. H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [59] D. C. Schmidt., and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine, Special Issue on Design Patterns, April, 1999.
- [60] M. Román, F. Kon and R. Campbell, "Reflective Middleware: From Your Desk to Your Hand", IEEE Distributed Systems Online, Vol. 2, No. 5, July 2001, http://dsonline.computer.org/0105/features/rom0105_print.htm
- [61] B. N. Jørgensen, E. Truyen, F. Matthijs and W. Joosen, "Customization of Object Request Brokers by Application Specific Policies". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [62] E. Truyen, B. N. Jørgensen and W. Joosen, "Customization of Component-based Object Request Brokers through Dynamic Reconfiguration", In: Proceedings of TOOLS EUROPE 2000. IEEE Computer Society, 2000, pp 181-194.
- [63] Sun Microsystems, JavaBeans Specification v1.0.1, August 1997, <http://java.sun.com/products/javabeans/index.jsp>
- [64] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software", In Proceedings of Reflection 2001, LNCS 2192.
- [65] F. Siqueira and V. Cahill. "Quartz: A QoS Architecture for Open Systems". In Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS 2000).
- [66] J. A. Zinky, D. E. Bakken, R. Schantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems, Volume 3, Number 1, January 1997, pp 55-73.
- [67] N. Wang, D. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall and R. Schantz, " Quality of Service Provisioning in Middleware and Applications", Microprocessors and Microsystems, special issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet (Paolo Bellavista ed.), vol. 27, no. 2, , March 2003, pp. 45-54.
- [68] A. Ulbrich, T. Weis, K. Geihs and C. Becker, "DotQoS - A QoS Extension for .NET Remoting", International Workshop on Quality of Service (IWQoS 2003), LNCS, Vol. 2707, Berkeley, CA, USA, June 2-4, 2003, pp 363-380.
- [69] T. Kristensen and T. Plogemann, "Enabling Flexible QoS Support in the Object Request Broker COOL" , in Proceedings of International Workshop on Distributed Real-Time Systems (IWDRS 2000), in conjunction with the 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, Apr. 2000.
- [70] A. Charfi, M. Mezini, "Aspect-Oriented Web Service Composition with AO4BPEL", In Proc. of the European Conference on Web Services (ECOWS 2004), pp. 168-182, September 27-30, 2004, Erfurt, Germany, LNCS 3250, Springer-Verlag.
- [71] A. Charfi, M. Mezini, "Middleware Services for Web Service Compositions" Poster, in WWW2005, May 10-14, 2005, Chiba, Japan.
- [72] A. Charfi, M. Mezini, "Hybrid Web Service Composition: Business Processes Meet Business Rules", In Proc. of the Second International Conference on Service-Oriented Computing (ICSOC 2004), pp. 30-38, November 15-19, 2004, New York, USA.
- [73] A. Charfi, M. Mezini. "Using Aspects for Security Engineering of Web Service Compositions", In Proc. of ICWS 2005, July 12-15, 2005, Florida, U.S.A
- [74] A. G. Alonso, F. Casati, H. Kuno, V. Machiraju. Web Services: Concepts, Architectures, and Applications, Springer, 2004.
- [75] T. Cohen, J. Gil, "AspectJ2EE = AOP + J2EE. Towards an Aspect Based, Programmable and Extensible Middleware Framework", In Proc. of the 18th European Conference on Object-Oriented Programming (ECOOP 2004), June 14-18, 2004, Oslo, Norway.
- [76] J. Boner, "AspectWerkz 2: An Extensible Aspect Container". Online article. <http://www.theserverside.com/articles/article.tss?l=AspectWerkzP1>

- [77] J. Boner, “Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address it”, Invited Industry Talk Abstract Paper in AOSD 2004, March 22-26, 2004, Lancaster, UK.
- [78] J. Boner, “What are the key issues for commercial AOP use - how does AspectWerkz address them?”, In Proc. of the Third Aspect-Oriented Software Development Conference (AOSD'04), pp. 5-6, March 22-26, 2004, Lancaster, UK.
- [79] J. Boner, “AspectWerkz - dynamic AOP for Java Overview”, Online article, <http://aspectwerkz.codehaus.org/>, 2004.
- [80] M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya, “Separation of coordination in a dynamic aspect-oriented framework”, Proc. of the First International Conference on AOSD (AOSD 2002), pp. 134–140, April 22-26, Enschede, The Netherlands.
- [81] M. Pinto, L. Fuentes, J.M. Troya, “DAOP-ADL: An architecture description language for dynamic component and aspect-based development“, In Pfenning, F. and Smaragdakis, Y. (eds.), Proc. of the Second International Conference on GPCE, pp. 118–137, September 22-25, Erfurt, Germany, LNCS 2830, Springer-Verlag.
- [82] L. Fuentes, M. Pinto, P. Sánchez, “Dynamic Weaving in CAM/DAOP: An Application Architecture Driven Approach”, Proc. of the Dynamic Aspect Workshop in conjunction with AOSD 2005, March 14-18, Chicago, Illinois.
- [83] M. Pinto, L. Fuentes, J.M. Troya, “A Dynamic Component and Aspect Platform”, accepted for publication in The Computer Journal.
- [84] R. Pawlak, L. Duchien, L. Seinturier, F. Legond-Aubry, G. Florin and L. Martelli, “JAC: An Aspect-based Distributed Dynamic Framework”, Journal Software Practice and Experience (SPE). 34(12):1119-1148. October 2004.
- [85] R. Pawlak, L. Seinturier, L. Duchien and G. Florin, “JAC: A Flexible Framework for AOP in Java”, Reflection'01, pp 1-24, September 2001, Kyoto, Japan. LNCS 2192, Springer Verlag.
- [86] R. Pawlak, L. Seinturier, L. Duchien and G. Florin, “Dynamic wrappers: handling the composition issue with JAC”, In Proc. of TOOLS-USA 2001, pp. 56-65, August 2001, IEEE, Santa-Barbara, USA.
- [87] JBoss Online User Guide, <http://docs.jboss.org/aop/aspect-framework/userguide/en/html/index.html>
- [88] The Aspect Oriented Programming and JBoss tutorial, http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html
- [89] The JBoss Application Server, <http://www.jboss.com/products/overview/jbossas>
- [90] E. Truyen, “Dynamic and Context-Sensitive Composition in Distributed Systems”, PhD Thesis, K.U.Leuven, Belgium, November 2004
- [91] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, B. N. Joergensen, “Dynamic and Selective Combination of Extensions in Component-based Applications”, In Proc. Of the 23rd International Conference on Software Engineering (ICSE'2001), May 2001, Toronto, Canada.
- [92] N. Ali, J. Pérez, C. Costa, J.A. Carsí, I. Ramos, “Implementation of the PRISMA Model in the .Net Platform”, In Proc. of DYNAMICA, Málaga, 2004.
- [93] J. Pérez, I. Ramos, J. Jaén, P. Letelier, “PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures”, In Proc. of the International Conference on Quality Software, pp. 59-66, 6-7 November, 2003, Dallas, Texas.
- [94] N. Ali, Josep S. Galiana, J. Jaén, I. Ramos, J.A. Carsí, J. Pérez, “Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures“, In Proc. of the 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2003), 3-5 November, Marina del Rey, CA, USA.
- [95] A. Popovici, T. Gross, G. Alonso. Dynamic weaving for aspect-oriented programming. Proc. of the First International Conference on AOSD, pp. 141-147, Enschede, The Netherlands, ACM Press, April 2002.
- [96] A. Popovici, G. Alonso, T. Gross, “Just in Time Aspects: Efficient Dynamic Weaving for Java”, In Proc. of the 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 2003.
- [97] A. Frei, A. Popovici, G. Alonso, “Event based systems as adaptive middleware platforms”. Workshop of the 17th European Conference for Object-Oriented Programming, Darmstadt, Germany July 2003.
- [98] A. Popovici, G. Alonso, T. Gross, “Spontaneous Container Services”, In Proc. of the 17th European Conference for Object-Oriented Programming, Darmstadt, Germany, July 2003.
- [99] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, D. Davison, D. Kopylenko, T. Risberg, M. Pollack, R. Harro, “Spring- Java/J2EE Application Framework”. Reference Documentation <http://www.springframework.org/docs/reference/index.html>

- [100] R. Johnson, "Expert One-on-One J2EE Design and Development by Rod Johnson". Ed. Wrox, 2002.
- [101] B. Verheecke, M.A. Cibrán, W. Vanderperren, D. Suvee, V. Jonckers, "AOP for Dynamic Configuration and Management of Web services in Client-Applications", International Journal on Web Services Research (JWSR): Volume 1, Issue 3, July-Sept 2004
- [102] B. Verheecke, M.A. Cibrán M.A, V Jonckers, "Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection", In Proc. of the European Conference on Web Services 2004 (ECOWS'04), Erfurt, Germany, September 2004.
- [103] B. Verheecke and M. A. Cibrán, "AOP for Dynamic Configuration and Management of Web services in Client-Applications", Published in the Proceedings of 2003 International Conference on Web Services - Europe (ICWS'03-Europe), Erfurt (Germany), September 2003
- [104] D. Suvee, W. Vanderperren, V. Jonckers. "JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development", In Proceedings of international conference on aspect-oriented software development (AOSD), Boston, USA, ACM Press, March 2003, pp 21-29.
- [105] D. Verspecht, W. Vanderperren, D. Suvee and V. Jonckers, "JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services", In Proceedings of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. Published in Mathematical modelling in Physics, Engineering and Cognitive Sciences, Vol. 8, November 2003.
- [106] Y. Coady and G. Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," Proceedings 2nd International Conference on Aspect-Oriented Software Development, pp. 50-59, 2003.
- [107] Y. Coady, G. Kiczales, M. Feeley, and J. S. Hutchinson, "Structuring Operating System Aspects," Communications of the ACM, vol. 10, pp. 79-82, 2001.
- [108] Y. Coady, G. Kiczales, M. Feeley, and S. G., "Using AspectC to Improve the Modularity of Path-Specific Customization in Operation System Code," Proceedings Joint ESEC and FSE-9, 2001.
- [109] C. Gibbs and Y. Coady, "OASIS: Organic Aspects for System Infrastructure Software - Easing evolution and adaptation through natural decomposition," Proceedings ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.
- [110] S. Bray, M. Yuen, Y. Coady, and M. E. Fiuczynski, "Managing Variability in Systems: Oh What a Tangled OS We Weave," Proceedings Workshop on Managing Variabilities Consistently in Design and Code, OOPSLA 2004, 2004.
- [111] M. Engel and B. Freislenben, "Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects," Proceedings of the Fourth International Conference on AOSD, 2005.
- [112] M. Engel and B. Freislenben, "Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels," Proceedings ACP4IS Workshop, AOSD 2005, 2005.
- [113] R. Douence, T. Fritz, N. Lorient, J.-M. Menuad, M. Segura-Devillechaise, and M. Sudholt, "An expressive aspect language for system applications with Arachne," Proceedings 4th International Conference on AOSD, 2005.
- [114] R. Aberg, J. L. Lawall, M. Südholt, G. Muller, A.-F. Le Meur, "On the automatic evolution of an OS kernel using temporal logic and AOP", Int. Conf. on Automated Software Engineering (ASE'03), IEEE, Oct. 2003.
- [115] G. Muller, J. L. Lawall, J.-M. Menaud, M. Südholt, "Constructing Component-Based Extension Interfaces in Legacy Systems Code", 11th ACM SIGOPS European Workshop, Sep. 2004.
- [116] F. Hunleth and R. Cytron, "Footprint and Feature Management Using Aspect Oriented Programming Techniques," presented at LCTES 02, Berlin, Germany, 2002.
- [117] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," presented at OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida, 2001.
- [118] A. Rashid and P. Sawyer, "Aspect-orientation and database systems: an effective customisation approach," IEE Proceedings - Software, vol. 148, pp. 156-164, 2001.
- [119] E. Wohlstader, S. Jackson, and P. Devanbu, "DADO: Enhancing Middleware to support crosscutting features in distributed, heterogeneous systems," Proceedings 25th International Conference on Software Engineering, pp. 174.
- [120] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," Proceedings 2nd International Conference on Aspect-Oriented Software Development, pp. 130-139, 2003.
- [121] C. Zhang and H.-A. Jacobsen, "Re-factoring Middleware Systems: A Case Study," Distributed Objects and Applications (DOA), 2003.

- [122] C. Zhang and H.-A. Jacobsen, "Resolving Feature Convolution using Horizontal Decomposition in Middleware Systems," University of Toronto, Computer Systems Research Group Technical Report Nr: 475, 2003.
- [123] C. Zhang and H.-A. Jacobsen, "Resolving feature convolution in middleware systems," Proceedings OOPSLA 2004, pp. 188-205, 2004.
- [124] M. Eichberg and M. Mezini, "Alice: Modularization of Middleware using Aspect-Oriented Programming," Software Engineering and Middleware (SEM), 2004.
- [125] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for Component Integration in Middleware," Practitioner Report, OOPSLA 2003, 2003.
- [126] A. Colyer and A. Clement, "Large Scale AOSD for Middleware," Proceedings 3rd International Conference on AOSD, 2004.
- [127] Pierre-Charles David and Thomas Ledoux: "Towards a Framework for Self-Adaptive Component-Based Applications", Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors, Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003, volume 2893 of Lecture Notes in Computer Science, Paris, pages 1--14, November 2003. Federated Conferences, Springer-Verlag.
- [128] M. Segura-Devillechaise, J.-M. Menuad, G. Muller, and J. Lawall, "Web Cache Prefetching as an aspect: Towards a Dynamic-Weaving Based Solution," Proceedings 2nd International Conference on Aspect-Oriented Software Development, 2003.
- [129] A. Rashid and R. Chitchyan, "Persistence as an Aspect," Proceedings 2nd International Conference on Aspect-Oriented Software Development, 2003.
- [130] A. Schmidmeier, "Using AspectJ to Eliminate Tangling code in EAI Activities," Practitioner Report, AOSD 2003, 2003.
- [131] J. van Gorp and J. Bosch, "Design Erosion: Problems & Causes", Journal of Systems & Software, vol 61, issue 2, 2002.
- [132] B. De Fraine, W. Vanderperren, D. Suvee, and J. Brichau, "Jumping Aspects Revisited", In Proceedings of DAW 2005, Chicago, USA, March 2005
- [133] Business Process Execution Language for Web Services home page <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>
- [134] BEA Systems home page, <http://www.bea.com>
- [135] The Jakarta project home page, <http://jakarta.apache.org/tomcat>
- [136] IBM Websphere home page, <http://www-306.ibm.com/software/websphere>
- [137] AspectJ home page, <http://eclipse.org/aspectj>
- [138] R. Pawlak, "The AOP Alliance: Why Did We Get In?", White Paper draft 2003.
- [139] Maven home page, <http://maven.apache.org/>
- [140] Netbeans home page, <http://www.netbeans.org/>
- [141] Eclipse home page, <http://www.eclipse.org>
- [142] Java Management Extensions home page, <http://java.sun.com/products/JavaManagement>
- [143] W. Joosen, F. Matthijs, J. Van Oeyen, B. Robben, S. Bijnens and P. Verbaeten, "CORRELATE: High-level Support for Traveling Agents". Technical Report CW236, dept. of Computer Science, K.U.Leuven, October 1996.
- [144] Hibernate home page, <http://www.hibernate.org>
- [145] Java Data Objects home page, <http://java.sun.com/products/jdo>
- [146] iBATIS home page, <http://www.ibatis.com>