2

Adaptive and Reflective Middleware

Edward Curry
National University of Ireland, Galway, Ireland

2.1 Introduction

Middleware platforms and related services form a vital cog in the construction of robust distributed systems. Middleware facilitates the development of large software systems by relieving the burden on the applications developer of writing a number of complex infrastructure services needed by the system; these services include persistence, distribution, transactions, load balancing, clustering, and so on.

The demands of future computing environments will require a more flexible system infrastructure that can adapt to dynamic changes in application requirements and environmental conditions. Next-generation systems will require predictable behavior in areas such as throughput, scalability, dependability, and security. This increase in complexity of an already complex software development process will only add to the already high rates of project failure.

Middleware platforms have traditionally been designed as monolithic static systems. The vigorous dynamic demands of future environments such as large-scale distribution or ubiquitous and pervasive computing will require extreme scaling into large, small, and mobile environments. In order to meet the challenges presented in such environments, next-generation middleware researchers are developing techniques to enable middleware platforms to obtain information concerning environmental conditions and adapt their behavior to better serve their current deployment. Such capability will be a prerequisite for any next-generation middleware; research to date has exposed a number of promising techniques that give middleware the ability to meet these challenges head on.

Adaptive and reflective techniques have been noted as a key emerging paradigm for the development of dynamic next-generation middleware platforms [1, 2]. These

Middleware for Communications. Edited by Qusay H. Mahmoud
 ISBN 0-470-86206-8

techniques empower a system to automatically self-alter (adapt) to meet its environment and user needs. Adaptive and Reflective system support advanced adaptive behavior. Adaptation can take place autonomously or semiautonomously, on the basis of the systems deployment environment, or within the defined policies of users or administrators [3].

The objective of this chapter is to explore adaptive and reflective techniques, their motivation for use, and introduce their fundamental concepts. The application of these techniques will be examined, and a summary of a selection of middleware platforms that utilize these techniques will be conducted. The tools and techniques that allow a system to alter its behavior will be examined; these methods are vital to implementing adaptive and reflective systems. Potential future directions for research will be highlighted; these include advances in programming techniques, open research issues, and the relationship to autonomic computing systems.

2.1.1 Adaptive Middleware

Traditionally, middleware platforms are designed for a particular application domain or deployment scenario. In reality, multiple domains overlap and deployment environments are dynamic, not static; current middleware technology does not provide support for coping with such conditions. Present research has been focused on investigating the possibility of enabling middleware to serve multiple domains and deployment environments. In recent years, platforms have emerged that support reconfigurability, allowing platforms to be customized for a specific task; this work has led to the development of adaptive multipurpose middleware platforms.

Adapt–a. To alter or modify so as to fit for a new use

An adaptive system has the ability to change its behavior and functionality. Adaptive middleware is software whose functional behavior can be modified dynamically to optimize for a change in environmental conditions or requirements [4]. These adaptations can be triggered by changes made to a configuration file by an administrator, by instructions from another program, or by requests from its users. The primary requirements of a runtime adaptive system are *measurement*, *reporting*, *control*, *feedback*, and *stability* [1].

2.1.2 Reflective Middleware

The groundbreaking work on reflective programming was carried out by Brian Smith at MIT [5]. Reflective middleware is the next logical step once an adaptive middleware has been achieved. A reflective system is one that can examine and reason about its capabilities and operating environment, allowing it to self-adapt at runtime. Reflective middleware builds on adaptive middleware by providing the means to allow the internals of a system to be manipulated and adapted at runtime; this approach allows for the automated self-examination of systems capabilities and the automated adjustment and optimization of those capabilities. The process of self-adaptation allows a system to provide an improved service for its environment or user's needs. Reflective platforms support advanced adaptive behavior; adaptation can take place autonomously on the basis of the status of the systems, environment, or in the defined policies of its users or administrators [3].

Reflect–v. To turn (back), cast (the eye or thought) on or upon something

Reflection is currently a hot research topic within software engineering and development. A common definition of reflection is a system that provides a representation of its own behavior that is amenable to inspection and adaptation and is causally connected to the underlying behavior it describes [6]. Reflective research is also gaining speed within the middleware research community. The use of reflection within middleware for advanced adaptive behavior gives middleware developers the tools to meet the challenges of next-generation middleware, and its use in this capacity has been advocated by a number of leading middleware researchers [1, 7].

Reflective middleware is self-aware middleware [8]

The reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports the development of flexible and adaptive systems and applications [8]. This reflective flexibility diminishes the importance of many initial design decisions by offering late-binding and runtime-binding options to accommodate actual operating environments at the time of deployment, instead of only anticipated operating environments at design time. [1]

A common definition of a reflective system [6] is a system that has the following:

Self-Representation: A description of its own behavior
Causally Connected: Alterations made to the self-representation are mirrored in the system's actual state and behavior

Causally Connected Self Representation (CCSR)

Few aspects of a middleware platform would not benefit from the use of reflective techniques. Research is ongoing into the application of these techniques in a number of areas within middleware platforms. While still relatively new, reflective techniques have already been applied to a number of nonfunctional areas of middleware. One of the main reasons nonfunctional system properties are popular candidates for reflection is the ease and flexibility of their configuration and reconfiguration during runtime, and changes to a nonfunctional system property will not directly interfere with a systems user interaction protocols. Nonfunctional system properties that have been enhanced with adaptive and reflective techniques include distribution, responsiveness, availability, reliability, fault-tolerance, scalability, transactions, and security.

Two main forms of reflection exist, behavioral and structural reflection. Behavioral reflection is the ability to intercept an operation and alter its behavior. Structural reflection is the ability to alter the programmatic definition of a programs structure. Low-level structural reflection is most commonly found in programming languages, that is, to change the definition of a class, a function, or a data structure on demand is outside the scope of this chapter. In this chapter, the focus is on behavioral reflection, specifically altering the behavior of middleware platforms at runtime, and structural reflection concerned with the high-level system architecture and selection of plugable service implementations used in a middleware platform.

2.1.3 Are Adaptive and Reflective Techniques the Same?

Adaptive and Reflective techniques are intimately related, but have distinct differences and individual characteristics:

—An adaptive system is capable of changing its behavior.
—A reflective system can inspect/examine its internal state and environment.

Systems can be both adaptive and reflective, can be adaptive but not reflective, as well as reflective but not adaptive. On their own, both of these techniques are useful, but when used collectively, they provide a very powerful paradigm that allows for system inspection with an appropriate behavior adaptation if needed. When talking about reflective systems, it is often assumed that the system has adaptive capabilities.

Common Terms

Reification
The process of providing an external representation of the internals of a system. This representation allows for the internals of the system to be manipulated at runtime.

Absorption
This is the process of enacting the changes made to the external representation of the system back into the internal system. Absorbing these changes into the system realizes the casual connection between the model and system.

Structural Reflection
Structural reflection provides the ability to alter the statically fixed internal data/functional structures and architecture used in a program. A structural reflective system would provide a complete reification of its internal methods and state, allowing them to be inspected and changed. For example, the definition of a class, a method, a function, and so on, may be altered on demand.
Structural reflection changes the internal makeup of a program

Behavioral Reflection
Behavioral reflection is the ability to intercept an operation such as a method invocation and alter the behavior of that operation. This allows a program, or another program, to change the way it functions and behaves.
Behavioral reflection alters the actions of a program

Nonfunctional Properties
The nonfunctional properties of a system are the behaviors of the system that are not obvious or visible from interaction with the system. Nonfunctional properties include distribution, responsiveness, availability, reliability, scalability, transactions, and security.

2.1.4 Triggers of Adaptive and Reflective Behavior

In essence, the reflective capabilities of a system should trigger the adaptive capabilities of a system. However, what exactly can be inspected in order to trigger an appropriate adaptive behavior? Typically, a number of areas within a middleware platform, its functionality, and its environment are amenable to inspection, measurement, and reasoning as to the optimum or desired performance/functionality. Software components known as *interceptors* can be inserted into the execution path of a system to monitor its actions. Using interceptors and similar techniques, reflective systems can extract useful information from the current execution environment and perform an analysis on this information.

Usually, a reflective system will have a number of interceptors and system monitors that can be used to examine the state of a system, reporting system information such as its performance, workload, or current resource usage. On the basis of an analysis of this information, appropriate alterations may be made to the system behavior. Potential monitoring tools and feedback mechanisms include performance graphs, benchmarking, user usage patterns, and changes to the physical deployments infrastructure of a platform (network bandwidth, hardware systems, etc).

2.2 Implementation Techniques

Software development has evolved from the 'on-the-metal' programming of assembly and machine codes to higher-level paradigms such as procedural, structured, functional, logic, and Object-Orientation. Each of these paradigms has provided new tools and techniques to facilitate the creation of complex software systems with speed, ease, and at lower development costs.

In addition to advancements in programming languages and paradigms, a number of techniques have been developed that allow flexible dynamic systems to be created. These techniques are used in adaptive systems to enable their behavior and functionality changes. This section provides an overview of such techniques, including meta-level programming, components and component framework, generative programming, and aspect-oriented programming.

2.2.1 Meta-Level Programming

In 1991, Gregor Kiczale's work on combining the concept of computational reflection and object-oriented programming techniques lead to the definition of a meta-object protocol [9]. One of the key aspects of this groundbreaking work was in the separation of a system into two levels. The base-level provides system functionality, and the meta-level contains the policies and strategies for the behavior of the system. The inspection and alteration of this meta-level allows for changes in the system's behavior.

The base-level provides the implementation of the system and exposes a meta-interface that can be accessed at the meta-level. This meta-interface exposes the internals of the base-level components/objects, allowing it to be examined and its behavior to be altered and reconfigured. The base-level can now be reconfigured to maximize and fine-tune the systems characteristics and behavior to improve performance in different contexts and operational environments. This is often referred to as the *Meta-Object Protocol* or MOP.

The design of a meta-interface/MOP is central to studies of reflection, and the interface should be sufficiently general to permit unanticipated changes to the platform but should also be restricted to prevent the integrity of the system from being destroyed [10].

Meta Terms Explained

Meta-
Prefixed to technical terms to denote software, data, and so on, which operate at a higher level of abstraction – Oxford English Dictionary

Meta-Level
The level of software that abstracts the functional and structural level of a system. Meta-level architectures are systems designed with a base-level (implementation level) that handles the execution of services and operations, and a meta-level that provides an abstraction of the base-level.

Meta-Object
The participants in an object-oriented meta-level are known as meta-objects

Meta-Object Protocol
The protocol used to communicate with the meta-object is known as the Meta-Object Protocol (MOP)

2.2.2 Software Components and Frameworks

With increasing complexity in system requirements and tight development budget constraints, the process of programming applications from scratch is becoming less feasible. Constructing applications from a collection of reusable components and frameworks is emerging as a popular approach to software development.

A software component is a functional discrete block of logic. Components can be full applications or encapsulated functionality that can be used as part of a larger application, enabling the construction of applications using components as building blocks. Components have a number of benefits as they simplify application development and maintenance, allowing systems to be more adaptive and respond rapidly to changing requirements. Reusable components are designed to encompass a reusable block of software, logic, or functionality. In recent years, there is increased interest in the use of components as a mechanism of building middleware platforms; this approach has enabled middleware platforms to be highly flexible to changing requirements.

Component frameworks are a collection of interfaces and interaction protocols that define how components interact with each other and the framework itself, in essence frameworks allow components to be plugged into them. Examples of component frameworks include Enterprise Java Beans (EJB) [11] developed by Sun Microsystems, Microsoft's .NET [12] and the CORBA Component Model (CMM) [13]. Components frameworks have also been used as a medium for components to access middleware services, for example, the EJB component model simplifies the development of middleware applications by providing automatic support for services such as transactions, security, clustering, database connectivity, life-cycle management, instance pooling, and so on. If

components are analogous to building blocks, frameworks can be seen as the cement that holds them together.

The component-oriented development paradigm is seen as a major milestone in software construction techniques. The process of creating applications by composing preconstructed program 'blocks' can drastically reduce the cost of software development. Components and component frameworks leverage previous• development efforts by capturing key implementation patterns, allowing their reuse in future systems. In addition, the use of replaceable software components can improve reliability, simplify the implementation, and reduce the maintenance of complex applications [14].

• Q1

2.2.3 Generative Programming

Generative programming [15] is the process of creating programs that construct other programs. The basic objective of a generative program, also known as a program generator [16], is to automate the tedious and error-prone tasks of programming. Given a requirements specification, a highly customized and optimized application can be automatically manufactured on demand. Program generators manufacture source code in a target language from a program specification expressed in a higher-level Domain Specific Language (DSL). Once the requirements of the system are defined in the higher-level DSL, the target language used to implement the system may be changed. For example, given the specification of text file format, a program generator could be used to create a driver program to edit files in this specified format. The program generator could use Java, C, Visual Basic (VB) or any other language as the target language for implementation; two program generators could be created, a Java version and a C version. This would allow the user a choice for the implementation of the driver program.

Generative programming allows for high levels of code reuse in systems that share common concepts and tasks, providing an effective method of supporting multiple variants of a program; this collection of variants is known as a *program family*. Program generation techniques may also be used to create systems capable of adaptive behavior via program recompilation.

2.3 Overview of Current Research

Adaptive and reflective capabilities will be commonplace in future next-generation middleware platforms. There is consensus [1, 8] that middleware technologies will continue to incorporate this new functionality. At present, these techniques have been applied to a number of middleware areas. There is a growing interest in developing reflective middleware with a large number of researchers and research groups carrying out investigations in this area. A number of systems have been developed that employ adaptive and reflective techniques, this section provides an overview of some of the more popular systems to have emerged.

2.3.1 Reflective and Adaptive Middleware Workshops

Reflective and adaptive middleware is a very active research field with the completion of a successful workshop on the subject at the IFIP/ACM Middleware 2000 conference [17].

Papers presented at this workshop cover a number of topics including reflective and adaptive architectures and systems, mathematical model and performance measurements of reflective platforms. Building on the success of this event a second workshop took place at Middleware 2003, The 2^{nd} Workshop on Reflective and Adaptive Middleware [18] covered a number of topics including nonfunctional properties, distribution, components, and future research trends.

2.3.2 Nonfunctional Properties

Nonfunctional properties of middleware platforms have proved to be very popular candidates for enhancement with adaptive and reflective techniques. These system properties are the behaviors of the system that are not obvious or visible from interaction with the system. This is one of the primary reasons they have proved popular with researchers, because they are not visible in user/system interactions and changes made to these properties will not affect the user/system interaction protocol. Nonfunctional properties that have been enhanced with adaptive and reflective techniques include distribution, responsiveness, availability, reliability, scalability, transactions, and security.

2.3.2.1 Security

The specialized Obol [19] programming language provides flexible security mechanisms for the Open ORB Python Prototype (OOPP). In OOPP, the flexible security mechanisms based on Obol is a subset of the reflective features of the middleware platform enabling programmable security via Obol. Reflective techniques within OOPP provide the mechanisms needed to access and modify the environment; Obol is able to access the environment meta-model making it possible to change and replace security protocols without changing the implementation of the components or middleware platform.

2.3.2.2 Quality-of-Service

A system with a Quality-of-Service (QoS) demand is one that will perform unacceptably if it is not carefully configured and tuned for the anticipated environment and deployment infrastructure. Systems may provide different levels of service to the end-user, depending on the deployment environment and operational conditions. An application that is targeted to perform well in a specific deployment environment will most likely have trouble if the environment changes. As an illustration of this concept, imagine a system designed to support 100 simultaneous concurrent users, if the system was deployed in an environment with 1000 or 10,000 users it will most likely struggle to provide the same level of service or QoS when faced with demands that are 10 or 100 times greater than what it is designed to handle.

Another example is a mobile distributed multimedia application. This type of application may experience drastic changes in the amount of bandwidth provided by the underlying network infrastructure from the broadband connections offered by residential or office networks to the 9600 bps GSM connection used while traveling. An application designed to operate on a broadband network will encounter serious difficulties when deployed over the substantially slower GSM-based connection. Researchers at Lancaster University have developed a reflective middleware platform [10] that adapts to the underlying network

infrastructure in order to improve the QoS provided by the application. This research alters the methods used to deliver the content to the mobile client, achieved by using an appropriate video and audio compression component for the network bandwidth available or the addition of a jitter-smoothing buffer to a network with erratic delay characteristics.

2.3.2.3 Fault-Tolerant Components

Adaptive Fault-Tolerance in the CORBA Component Model (AFT-CCM) [20] is designed for building component-based applications with QoS requirements related to fault-tolerance. AFT-CCM is based on the CORBA Component Model (CCM) [13] and allows an application user to specify QoS requirements such as levels of dependability or availability for a component. On the basis of these requirements, an appropriate replication technique and the quantity of component replicas will be set to achieve the target. These techniques allow a component-based distributed application to be tolerant of possible component and machine faults. The AFT-CCM model enables fault-tolerance in a component with complete transparency for the application without requiring changes to its implementation.

2.3.3 Distribution Mechanism

A number of reflective research projects focus on improving the flexibility of application distribution. This section examines the use of adaptive and reflective techniques in enhancing application distribution mechanisms.

2.3.3.1 GARF and CodA

Projects such as GARF and CodA are seen as a milestone in reflective research. GARF [21] (automatic generation of reliable applications) is an object-oriented tool that supports the design and programming of reliable distributed applications. GARF wraps the distribution primitives of a system to create a uniform abstract interface that allows the basic behavior of the system to be enhanced. One technique to improve application reliability is achieved by replicating the application's critical components over several machines. Group-communication schemes are used to implement these techniques by providing multicasting to deliver messages to groups of replicas. In order to implement this group-communication, multicasting functionality needs to be mixed with application functionally. GARF acts as an intermediate between group-communication functionality and applications; this promotes software modularity by clearly separating the implementation of concurrency, distribution, and replication from functional features of the application.

The CodA [22] project is a pioneering landmark in reflective research. Designed as an object meta-level architecture, its primary design goal was to allow for decomposition by logical behavior. Through the application of the decomposition OO technique, CodA eliminated the problems existing in 'monolithic' meta-architectures. CodA achieves this by using multiple meta-objects, with each one describing a single small behavioral aspect of an object, instead of using one large meta-object that describes all aspects of an objects behavior. Once the distribution concern has been wrapped in meta-objects, aspects of the systems distribution such as message queues, message sending, and receiving can be controlled. This approach offers a fine-grained approach to decomposition.

2.3.3.2 Reflective Architecture Framework for Distributed Applications

The Reflective Architecture Framework for Distributed Applications (RAFDA) [23] is a reflective framework enabling the transformation of a nondistributed application into a flexibly distributed equivalent one. RAFDA allows an application to adapt to its environment by dynamically altering its distribution boundaries. RAFDA can transform a local object into a remote object, and vice versa, allowing local and remote objects to be interchangeable.

As illustrated in Figure 2.1, RAFDA achieves flexible distribution boundaries by substituting an object with a proxy to a remote instance. In the above example, objects A and B both hold references to a shared instance of object C, all objects exist in a single address space (nondistributed). The objective is to move object C to a new address space. RAFDA transforms the application so that the instance of C is remote to its reference holders; the instance of C in address space A is replaced with a proxy, Cp, to the remote implementation of C in address space B.

The process of transformation is performed at the bytecode level. RAFDA identifies points of substitutability and extracts an interface for each substitutable class; every reference to a substitutable class must then be transformed to use the extracted interface. The proxy implementations provide a number of transport options including SOAP●, RMI,● and IIOP●. The use of interfaces makes nonremote and remote versions of a class interchangeable, thus allowing for flexible distribution boundaries. Policies determine substitutable classes and the transportation mechanisms used for the distribution.

● Q2
● Q3
● Q4

2.3.3.3 mChaRM

The Multi-Channel Reification Model (mChaRM) [24] is a reflective approach that reifies and reflects directly on communications. The mChaRM model does not operate on base-objects but on the communications between base-objects, resulting in a communication-oriented model of reflection. This approach abstracts and encapsulates interobject communications and enables the meta-programmer to enrich and or replace the predefined communication semantics. mChaRM handles a method call as a message sent through a logical channel between a set of senders and a set of receivers. The model supports the reification of such logical channels into logical objects called *multi-channels*. A multi-channel can enrich the messages (method calls) with new functionality. This

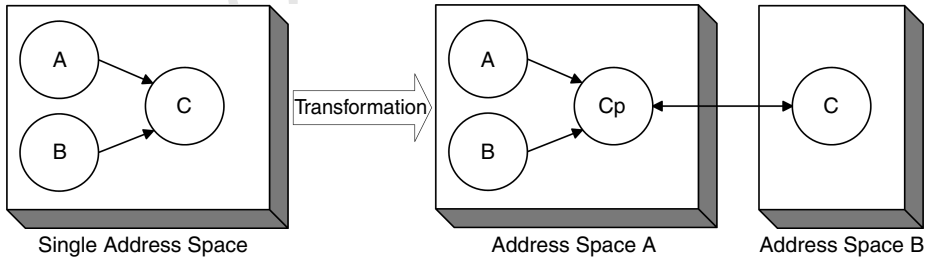

Figure 2.1 RAFDA redistribution transformation. Reproduced by permission of Springer, in Portillo, A. R., Walker, S., Kirby, G., *et al.* (2003) A Reflective Approach to Providing Flexibility in Application Distribution. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil

technique allows for a finer reification reflection granularity than those used in previous approaches, and for a simplified approach to the development of communication-oriented software. mChaRM is specifically targeted for designing and developing complex communication mechanism from the ground up, or for extending the behavior of a current communication mechanism; it has been used to extend the standard Java RMI framework to one supporting multicast RMI.

2.3.3.4 Open ORB

The Common Object Request Broker Architecture (CORBA) [25] is a popular choice for research projects applying adaptive and reflective techniques. A number of projects have incorporated these techniques into CORBA Object Request Brokers or ORBs.

The Open ORB 2 [10] is an adaptive and dynamically reconfigurable ORB supporting applications with dynamic requirements. Open ORB has been designed from the ground up to be consistent with the principles of reflection. Open ORB exposes an interface (framework) that allows components to be plugable; these components control several aspects of the ORBs behavior including thread and buffer management and protocols. Open ORB is implemented as a collection of configurable components that can be selected at build-time and reconfigured at runtime; this process of component selection and configurability enables the ORB to be adaptive.

Open ORB is implemented with a clear separation between base-level and meta-level operations. The ORBs meta-level is a causally connected self-representation of the ORBs base-level (implementation) [10]. Each base-level component may have its own private set of meta-level components that are collectively referred to as the components meta-space. Open ORB has broken down its meta-space into several distinct models. The benefit of this approach is to simplify the interface to the meta-space by separating concerns between different system aspects, allowing each distinct meta-space model to give a different view of the platform implementation that can be independently reified. As shown in Figure 2.2, models cover the interfaces, architecture, interceptor, and resource

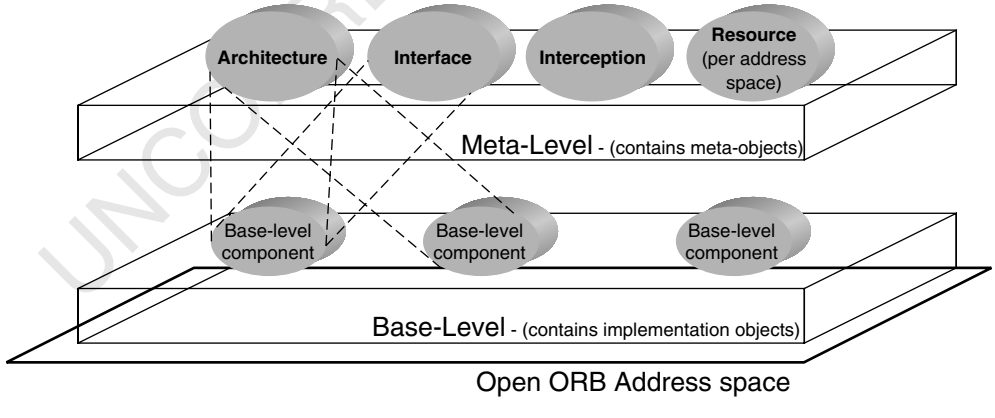

Figure 2.2 Open ORB architecture. Reproduced by permission of IEEE, in Blair, G. S., Coulson, G., Andersen, A., *et al.* (2001) The Design and Implementation of Open ORB 2, *IEEE Distributed Systems Online*, **2**(6)

meta-spaces. These models provide access to the underlying platform and component structure through reflection; every application-level component offers a meta-interface that provides access to an underlying meta-space, which is the support environment for the component.

Structural Reflection

Open ORB version 2 uses two meta-models to deal with structural reflection, one for its external interfaces and one for its internal architecture. The interface meta-model acts similar to the Java reflection API allowing for the dynamic discovery of a component's interfaces at runtime. The architecture meta-model details the implementation of a component broken down into two lines, a component graph (a local-binding of components) and an associated set of architectural constraints to prevent system instability [10]. Such an approach makes it possible to place strict controls on access rights for the ORBs adaptation. This allows all users the right to access the interface meta-model while restricting access rights to the architecture meta-model permitting only trusted third parties to modify the system architecture.

Behavioral Reflection

Two further meta-models exist for behavioral reflection, the interception, and resource models. The interception model enables the dynamic insertion of interceptors on a specific interface allowing for the addition of prebehavior and postbehavior. This technique may be used to introduce nonfunctional behavior into the ORB. Unique to Open ORB is its use of a resource meta-model allowing for access to the underlying system resources, including memory and threads, via resource abstraction, resource factories, and resource managers [10].

2.3.3.5 DynamicTAO – Real-Time CORBA

Another CORBA-based reflective middleware project is DynamicTAO [26]. DynamicTAO is designed to introduce dynamic reconfigurability into the TAO ORB [27] by adding reflective and adaptive capabilities. DynamicTAO enables on-the-fly reconfiguration and customization of the TAO ORBs internal engine, while ensuring it is maintained in a consistent state. The architecture of DynamicTAO is illustrated in Figure 2.3; in this architecture, reification is achieved through a collection of component configurators. Component implementations are provided via libraries. DynamicTAO allows these components to be dynamically loaded and unloaded from the ORBs process at runtime, enabling the ORB to be inspected and for its configuration to be adapted. Component implementations are organized into categories representing different aspects of the ORB's internal engine such as concurrency, security, monitoring, scheduling, and so on. Inspection in DynamicTAO is achieved through the use of interceptors that may be used to add support for monitoring, these interceptors may also be used to introduce behaviors for cryptography, compression, access control, and so on. DynamicTAO is designed to add reflective features to the TAO ORB, reusing the codebase of the existing TAO ORB results in a very flexible, dynamic, and customizable system implementation.

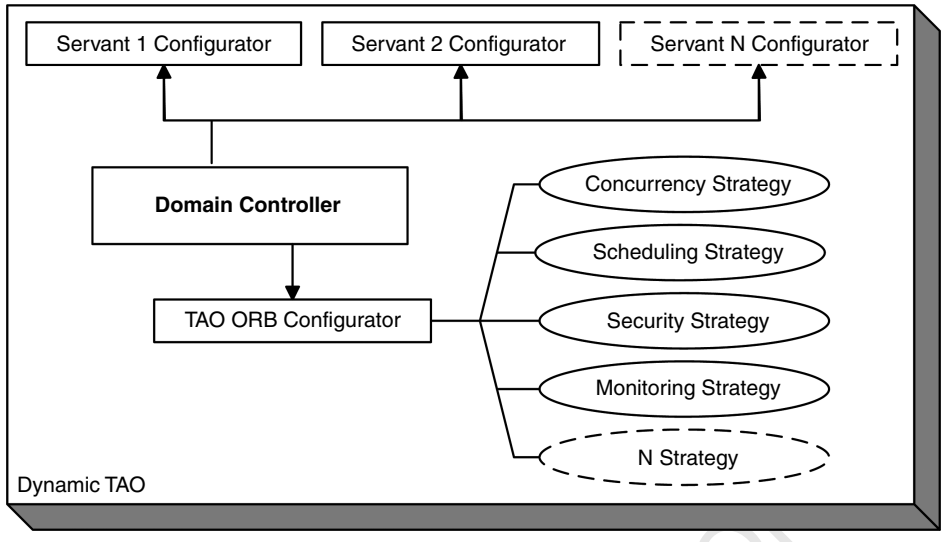

Figure 2.3 Architecture of dynamicTAO. Reproduced by permission of Springer, in Kon, F., Román, M., Liu, P., *et al.* (2002) Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York

2.3.3.6 Reflective Channel Hierarchies

This Chameleon messaging service [28] is a research prototype that provides a generic framework in which reflective techniques can perform customizations and adoptions to Message-Oriented Middleware (MOM). Chameleon focuses specifically on the application of such techniques within message queues and channel/topic hierarchies used as part of the publish/subscribe messaging model.

The publish/subscribe messaging model is a very powerful mechanism used to disseminate information between anonymous message consumers and producers. In the publish/subscribe model, clients producing messages "publish" these to a specific topic or channel. These channels are "subscribed" to by clients wishing to consume messages of interest to them. Hierarchical channel structures allows channels to be defined in a hierarchical fashion so that channels may be nested under other channels. Each subchannel offers a more granular selection of the messages contained in its parent channel. Clients of hierarchical channels subscribe to the most appropriate level of channel in order to receive the most relevant messages. For further information on the publish/subscribe, model and hierarchical channels, please refer to Chapter 1 (Message-Oriented Middleware).

Current MOM platforms do not define the structure of channel hierarchies. Application developers must therefore manually define the structure of the hierarchy at design time. This process can be tedious and error-prone. To solve this problem, the Chameleon messaging architecture implements reflective channel hierarchies [28] with the ability to autonomously self-adapt to their deployment environment. The Chameleon architecture exposes a causally connected meta-model to express the set-up and configuration of the

queues and the structure of the channel hierarchy, which enables the runtime inspection and adoption of the hierarchy and queues within MOM systems.

Chameleon's adaptive behavior originates from its reflection engine whose actions are governed by plugable reflective policies; these intelligent policies contain the rules and strategies used in the adaptation of the service. Potential policies could be designed to optimize the distribution mechanism for group messaging using IP multicasts or to provide support for federated messaging services using techniques from Hermes [29] or Herald [30]. Policies could also be designed to work with different levels of filtering (subject, content, composite) or to support different formats of message payloads (XML, JPEG, PDF, etc). Policies could also be used to introduce new behaviors into the service; potential behaviors include collaborative filtering/recommender systems, message transformations, monitoring and accounting functionality.

2.4 Future Research Directions

While it is difficult to forecast the future direction of any discipline, it is possible to highlight a number of developments on the immediate horizon that could affect the direction taken by reflective research over the coming years. This section will look at the implications of new software engineering techniques and highlight a number of open research issues and potential drawbacks that effect adaptive and reflective middleware platforms. The section will conclude with an introduction to autonomic computing and the potential synergy between autonomic and reflective systems.

2.4.1 Advances in Programming Techniques

The emergence of multifaceted software paradigms such as Aspect-Oriented Programming (AOP) and Multi-Dimensional Separation of Concerns (MDSOC) will have a profound effect on software construction. These new paradigms have a number of benefits for the application of adaptive and reflective techniques in middleware systems. This section provides a brief overview of these new programming techniques.

2.4.1.1 Aspect-Oriented Programming (AOP)

We can view a complex software system as a combined implementation of multiple concerns, including business-logic, performance, logging, data and state persistence, debugging and unit tests, error checking, multithreaded safety, security, and various other concerns. Most of these are system-wide concerns and are implemented throughout the entire codebase of the system; these system-wide concerns are known as crosscutting concerns.

The most popular practice for implementing adaptive and reflective systems is the Object-Oriented (OO) paradigm. Excluding the many benefits and advantages object-oriented programming has over other programming paradigms, object-oriented and reflective techniques have a natural fit. The OO paradigm is a major advancement in the way we think of and build software, but it is not a silver bullet and has a number of limitations. One of these limitations is the inadequate support for crosscutting concerns. The Aspect-Oriented-Programming (AOP) [31] methodology helps overcome this limitation,

AOP complements OO by creating another form of separation that allows the implementation of a crosscutting concern as a single unit. With this new method of concern separation, known as an *aspect*, crosscutting concerns are more straightforward to implement. Aspects can be changed, removed, or inserted into a systems codebase enabling the reusability of crosscutting code.

A brief illustration would be useful to explain the concept. The most commonly used example of a crosscutting concern is that of logging or execution tracking; this type of functionality is implemented throughout the entire codebase of an application making it difficult to change and maintain. AOP [31] allows this functionality to be implemented in a single aspect; this aspect can now be applied/weaved throughout the entire codebase to achieve the required functionality.

Dynamic AOP for Reflective Middleware

The Object-Oriented paradigm is widely used within reflective platforms. However, a clearer separation of crosscutting concerns would be of benefit to meta-level architectures. This provides the incentive to utilize AOP within reflective middleware projects.

A major impediment to the use of AOP techniques within reflective systems has been the implementation techniques used by the initial implementations of AOP [32]. Traditionally, when an aspect is inserted into an object, the compiler weaves the aspect into the objects code; this results in the absorption of the aspects into the object's runtime code. The lack of preservation of the aspects as an identifiable runtime entity is a hindrance to the dynamic adaptive capabilities of systems created with aspects. Workarounds to this problem exist in the form of dynamic system recompilation at runtime; however, this is not an ideal solution and a number of issues, such as the transference of the system state, pose problems.

Alternative implementations of AOP have emerged that do not have this limitation. These approaches propose an AOP method of middleware construction using composition [33] to preserve aspect as runtime entities, this method of creation facilitates the application of AOP for the construction of reflective middleware platforms. Another approach involving Java bytecode manipulation libraries such as Javassist [34] provide a promising method of implementing AOP frameworks (JBossAOP) with dynamic runtime aspect weaving.

One of the founding works on AOP highlighted the process of performance optimization that bloated a 768-line program to 35,213 lines. Rewriting the program with the use of AOP techniques reduced the code back to 1039 lines while retaining most of the performance benefits. Grady Booch, while discussing the future of software engineering techniques, predicts the rise of multifaceted software, that is, software that can be composed in multiple ways at once, he cites AOP as one of the first techniques to facilitate a multifaceted capability [35].

2.4.1.2 Multi-Dimensional Separation of Concerns

The key difference between AOP and Multi-Dimensional Separation of Concerns [36] (MDSOC) is the scale of multifaceted capabilities. AOP will allow multiple crosscutting aspects to be weaved into a program, thus changing its composition through the addition of these aspects. Unlike AOP, MDSOC multifaceted capabilities are not limited to the

use of aspects; MDSOC allows for the entire codebase to be multifaceted, enabling the software to be constructed in multiple dimensions.

MDSOC also supports the separation of concerns for a single model [37], when using AOP you start with base and use individually coded aspects to augment this base. Working from a specific base makes the development of the aspects more straightforward but also introduces limitations on the aspects, such as limitations on aspect composition [37]; you cannot have an aspect of an aspect. In addition, aspects can be tightly coupled to the codebase for which they are designed; this limits their reusability.

MDSOC enables software engineers to construct a collection of separate models, each encapsulating a concern within a class hierarchy specifically designed for that concern [37]. Each model can be understood in isolation, any model can be augmented in isolation, and any model can be augmented with another model. These techniques streamline the division of goals and tasks for developers. Even with these advances, the primary benefit of MDSOC comes from its ability to handle multiple decompositions of the same software simultaneously, some developers can work with classes, others with features, others with business rules, other with services, and so on, even though they model the system in substantially different ways [37].

To further illustrate these concepts, an example is needed, which by Ossher [37] is of a software company developing personnel management systems for large international organizations. For the sake of simplicity, assume that their software has two areas of functionality, personal tracking that records employees' personal details such as name, address, age, phone number, and so on, and payroll management that handles salary and tax information.

Different clients seeking similar software approach the fictitious company, they like the software but have specific requirements, some clients want the full system while others do not want the payroll functionality and refuse to put up with the extra overhead within their system implementation.

On the basis of market demands, the software house needs to be able to mix and match the payroll feature. It is extremely difficult to accomplish this sort of dynamic feature selection using standard object-oriented technology. MDSOC allows this flexibility to be achieved within the system with on-demand remodularization capabilities; it also allows the personnel and payroll functionality to be developed almost entirely separate using different class models that best suit the functionality they are implementing.

2.4.2 Open Research Issues

There are a number of open research issues with adaptive and reflective middleware systems. The resolution of these open issues is critical to the wide-scale deployment of adaptive and reflective techniques in production and mission critical environments. This section highlights a number of the more common issues.

2.4.2.1 Open Standards

The most important issue currently faced by adaptive and reflective middleware researchers is the development of an open standard for the interaction of their middleware platforms. An international consensus is needed on the interfaces and protocols used to interact with these platforms. The emergence of such standards is important to support the development

of next-generation middleware platforms that are configurable and reconfigurable and also to offer applications portability and interoperability across proprietary implementation of such platforms [10]. Service specification and standards are needed to provide a stable base upon which to create services for adaptive and reflective middleware platforms. Because of the large number of application domains that may use these techniques, one generic standard may not be enough; a number of standards may be needed.

As adaptive and reflective platforms mature, the ability of such system to dynamically discover components with corresponding configuration information at runtime would be desirable. Challenges exist with this proposition, while it is currently possible to examine a components interface at runtime; no clear method exists for documenting the functionality of neither a component nor its performance or behavioral characteristics. A standard specification is needed to specify what is offered by a component.

2.4.2.2 System Cooperation

One of the most interesting research challenges in future middleware platforms is the area of cooperation and coordination between middleware services to achieve a mutual beneficial outcome. Middleware platforms may provide different levels of services, depending on environmental conditions and resource availability and costs. John Donne said 'No man is an island'; likewise, no adaptive or reflective middleware platform, service, or component is an island, and each must be aware of both the individual consequences and group consequences of its actions. Next-generation middleware systems must coordinate/trade with each other in order to maximize the available resources to meet the system requirements.

To achieve this objective, a number of topics need to be investigated. The concept of negotiation-based adaptation will require mechanisms for the trading of resources and resource usage. A method of defining a resource, its capabilities, and an assurance of the QoS offered needs to be developed. Trading partners need to understand the commodities they are trading in. Resources may be traded in a number of ways from simple barter between two services to complex auctions with multiple participants, each with their own tradable resource budget, competing for the available resource. Once a trade has been finalized, enforceable contracts will be needed to ensure compliance with the trade agreement. This concept of resource trading could be used across organizational boundaries with the trading of unused or surplus resources in exchange for monetary reimbursement.

2.4.2.3 Resource Management

In order for next-generation middleware to maximize system resource usage and improve quality-of-service, it must have a greater knowledge with regard to the available resources and their current and projected status. Potentially, middleware platforms may wish to participate in system resource management. A number of resource support services will need to be developed including mechanisms to interact with a resource, obtain a resource's status, coordination techniques to allow a resource to be reserved for future usage at a specified time. A method to allow middleware to provide resource management policies to the underlying system-level resource managers or at the minimum to influence these policies by indicating the resources it will need to meet its requirements will also be required.

2.4.2.4 Performance

Adaptive and reflective systems may suffer in performance because of additional infrastructure required to facilitate adaptations and the extra self-inspection workload required by self-adaptation; such systems contain an additional performance overhead when compared to a traditional implementation of a similar system. However, under certain circumstances, the changes made to the platform through adaptations can improve performance and reduce the overall workload placed on the system. This saving achieved by adaptations may offset the performance overhead or even write it off completely.

System speed may not always be the most important measurement of performance for a given system, for example, the Java programming language is one of the most popular programming languages even though it is not the fastest language; other features such as its cross-platform compatibility make it a desirable option. With an adaptive and reflective platform, a performance decrease may be expected from the introduction of new features, what limits in performance are acceptable to pay for a new feature? What metrics may be used to measure such a trade-off? How can a real measurement of benefit be achieved?

Reflective systems will usually have a much larger codebase compared to a nonreflective one, which is due to the extra code needed to allow for the system to be inspected and adapted as well as the logic needed to evaluate and reason about the systems adaptation. This larger codebase results in the platform having a larger memory footprint. What techniques could be used to reduce this extra code? Could this code be made easily reusable within application domains?

2.4.2.5 Safe Adaptation

Reflection focuses on increasing flexibility and the level of openness. The lack of safe-bounds for preventing unconstrained system adaptation resulting in system malfunctions is a major concern for reflective middleware developers. This has been seen as an 'Achilles heel' of reflective systems [38]. It is important for system engineers to consider the impact that reflection may have on system integrity and to include relevant checks to ensure that integrity is maintained. Techniques such as architectural constraints are a step in the right direction to allowing safe adaptations. However, more research is needed in this area, particularly where dynamically discovered components are introduced into a platform. How do we ensure that such components will not corrupt the platform? How do we discover the existence of such problems? Again, standards will be needed to document component behavior with constant checking of its operations to ensure it does not stray from its contracted behavior.

2.4.2.6 Clearer Separation of Concerns

The clearer separation of concerns within code is an important issue for middleware platforms. A clear separation of concerns would reduce the work required to apply adaptive and reflective techniques to a larger number of areas within middleware systems. The use of dynamic AOP and MDSOC techniques to implement nonfunctional and crosscutting concerns eases the burden of introducing adaptive and reflective techniques within these areas.

The separation of concerns with respect to responsibility for adaptation is also an important research area, multiple subsystems within a platform may be competing for specific adaptations within the platform, and these adaptations may not be compatible with one another. With self-configuring systems and specifically when these systems evolve to become self-organizing groups, who is in charge of the group's behavior? Who performs the mediations between the conflicting systems? Who chooses what adaptations should take place? These issues are very important to the acceptance of a self-configuration system within production environments.

The Object Management Group (OMG) Model Driven Architecture (MDA) [39] defines an approach for developing systems that separates the specification of system functionality from the specification of the implementation of that functionality with a specific technology. MDA can be seen as an advance on the concept of generative programming. The MDA approach uses a Platform Independent Model (PIM) to express an abstract system design that can be implemented by mapping or transforming to one or more Platform Specific Models (PSMs). The major benefit of this approach is that you define a system model over a constantly changing implementation technology allowing your system to be easily updated to the latest technologies by simple switching to the PSMs for the new technology. The use of MDA in conjunction with reflective components-based middleware platforms could be a promising approach for developing future distributed systems.

2.4.2.7 Deployment into Production Environments

Deployment of adaptive and reflective systems into production mission critical environments will require these systems to reach a level of maturity where system administrators feel comfortable with such a platform in their environment. Of utmost importance to reaching this goal is the safe adaptation of the system with predictable results in the systems behavior. The current practices used to test systems are inadequate for adaptive and reflective systems. In order to be accepted as a deployable technology, it is important for the research community to develop the necessary practices and procedures to test adaptive and reflective systems to ensure they perform predictably, and such mechanisms will promote confidence in the technology. Adaptive and reflective techniques must also mature enough to require only the minimum amount of system adoption necessary to achieve the desired goal. Once these procedures are in place, an incremental approach to the deployment of these systems is needed; the safe coexistence of both technologies will be critical to acceptance, and it will be the responsibility of adaptive and reflective systems to ensure that their adaptations do not interfere with other systems that rely on them or systems they interact with.

2.4.3 Autonomic Computing

As system workloads and environments become more unpredictable and complex, they will require skilled administration personnel to install, configure, maintain, and provide 24/7 support. In order to solve this problem, IBM has announced an autonomic computing initiative. IBM's vision of autonomic computing [40] is an analogy with the human autonomic nervous system; this biological system relieves the conscious brain of the burden of having to deal with low-level routine bodily functions such as muscle use, cardiac muscle use (respiration), and glands. An autonomic computing system would relieve the burden

Table 2.1 Fundamental characteristics of autonomic systems

Characteristic	Description
Self-Configuring	The system must adapt automatically to its operating environment, hardware and software platforms must possess a self-representation of their abilities and to self-configure to the environment
Self-Healing	Systems must be able to diagnose and solve service interruptions. For a system to be self-healing, it must be able to recognize a failure and isolate it, thus shielding the rest of the system from its erroneous activity. It then must be capable of recovering transparently from failure by fixing or replacing the section of the system that is responsible for the error
Self-Optimizing	On a constant basis, the system must be evaluating potential optimizations. Through self-monitoring and resource tuning, and through self-configuration, the system should self-optimize to efficiently maximize resources to best meet the needs of its environment and end-user needs
Self-Protecting	Perhaps the most interesting of all the characteristics needed by an autonomic system is that self-protecting systems need to protect themselves from attack•. These systems must anticipate a potential attack, detect when an attack is under way, identify the type of attack, and use appropriate countermeasures to defeat or at least nullify the attack. Attacks on a system can be classified as Denial-of-Service (DoS) or the infiltration of an unauthorized user to sensitive information or system functionality

• Q5

of low-level functions such as installation, configuration, dependency management, performance optimization management, and routine maintenance from the conscious brain, the system administrators.

The basic goal of autonomic computing is to simplify and automate the management of computing systems, both hardware and software, allowing them to self-manage, without the need for human intervention. Four fundamental characteristics are needed by an autonomic system to be self-managing; these are described in Table 2.1. The common theme shared by all of these characteristics is that each of them requires the system to handle functionality that has been traditionally the responsibility of a human system administrator.

Within the software domain, adaptive and reflective techniques will play a key role in the construction of autonomic systems. Adaptive and reflective techniques already exhibit a number of the fundamental characteristics that are needed by autonomic systems. Thus, reflective and adaptive middleware provide the ideal foundations for the construction of autonomic middleware platforms. The merger of these two strands of research is a realistic prospect. The goals of autonomic computing highlight areas for the application of reflective and adaptive techniques, these areas include self-protection and self-healing, with some work already initiated in the area of fault-tolerance [20].

2.5 Summary

Middleware platforms are exposed to environments demanding the interoperability of heterogeneous systems, 24/7 reliability, high performance, scalability and security while

maintaining a high QoS. Traditional monolithic middleware platforms are capable of coping with such demands as they have been designed and fine-tuned in advance to meet these specific requirements. However, next-generation computing environments such as large-scale distribution, mobile, ubiquitous, and pervasive computing will present middleware with dynamic environments with constantly changing operating conditions, requirements, and underlying deployment infrastructures. Traditionally, static middleware platforms will struggle when exposed to these environments, thus providing the motivation to develop next-generation middleware systems to adequately service such environments.

To prepare next-generation middleware to cope with these scenarios, middleware researchers are developing techniques to allow middleware platforms to examine and reason about their environment. Middleware platforms can then self-adapt to suit the current operating conditions based on this analysis; such capability will be a prerequisite for next-generation middleware.

Two techniques have emerged that enable middleware to meet these challenges. Adaptive and reflective techniques allow applications to examine their environment and self-alter in response to dynamically changing environmental conditions, altering their behavior to service the current requirements. Adaptive and reflective middleware is a key emerging paradigm that will help simplify the development of dynamic next-generation middleware platforms [1, 2].

There is a growing interest in developing reflective middleware with a large number of researchers and research group's active in this area. Numerous architectures have been developed that employ adaptive and reflective techniques to allow for adaptive and self-adaptive capabilities; these techniques have been applied in a number of areas within middleware platforms including distribution, responsiveness, availability, reliability, concurrency, scalability, transactions, fault-tolerance, and security.

IBM's autonomic computing envisions a world of self-managing computer systems, and such autonomic systems will be capable of self-configuration, self-healing, self-optimization, and self-protection against attack, all without the need for human intervention. Adaptive and reflective enabled middleware platforms will play a key role in the construction of autonomic middleware as they share a number of common characteristics with autonomic systems.

Bibliography

[1] Schantz, R. E. and Schmidt, D. C. (2001) Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications, *Encyclopedia of Software Engineering*, Wiley & Sons.

[2] Geihs, K. (2001) Middleware Challenges Ahead. *IEEE Computer*, **34**(6).

[3] Blair, G. S., Costa, F. M., Coulson, G., *et al.* (1998) The Design of a Resource-Aware Reflective Middleware Architecture, *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, Springer, St. Malo, France.

[4] Loyall, J., Schantz, R., Zinky, J., *et al.* (2001) Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. *Proceedings of the 21st International Conference on Distributed Computing Systems*, Mesa, AZ.

[5] Smith, B. C. (1982) *Procedural Reflection in Programming Languages*, PhD Thesis, MIT Laboratory of Computer Science.

[6] Coulson, G. (2002) What is Reflective Middleware? *IEEE Distributed Systems Online*.●

[7] Geihs, K. (2001) Middleware Challenges Ahead. *IEEE Computer*, **34**(6).

[8] Kon, F., Costa, F., Blair, G., *et al.* (2002) The Case for Reflective Middleware. *Communications of the ACM*, **45**(6).

[9] Kiczales, G., Rivieres, J. d., and Bobrow, D. G. (1992) *The Art of the Metaobject Protocol*, MIT Press.●

[10] Blair, G. S., Coulson, G., Andersen, A., *et al.* (2001) The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, **2**(6).

[11] DeMichiel, L. G. and Sun Microsystems, Inc. *Enterprise JavaBeansTM Specification, Version 2.1*.●

[12] Microsoft.● *Overview of the .NET Framework White Paper*, http://msdn.microsoft.com

[13] Object Management Group (2002) *CORBA Components OMG Document formal/02-06-65*.

[14] Szyperski, C. (1997) *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley.

[15] Czarnecki, K. and Eisenecker, U. (2000) *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

[16] Cleaveland, C. (2001) *Program Generators with XML and Java*, Prentice Hall.

[17] Kon, F., Blair, G. S., and Campbell, R. H. (2000) Workshop on Reflective Middleware. *Proceedings of the IFIP/ACM Middleware 2000*, New York, USA.

[18] Corsaro, A., Wang, N., Venkatasubramanian, N., *et al.* (2003) The 2nd Workshop on Reflective and Adaptive Middleware. *Proceedings of the Middleware 2003*, Rio de Janeiro, Brazil.

[19] Andersen, A., Blair, G. S., Stabell-Kulo, T., *et al.* (2003) Reflective Middleware and Security: OOPP meets Obol. *Proceedings of the Workshop on Reflective Middleware, Middleware 2003*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[20] Favarim, F., Siqueira, F., and Fraga, J. (2003) Adaptive Fault-Tolerant CORBA Components. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil.

[21] Garbinato, B., Guerraoui, R., and Mazouni, K.R. (1993) Distributed Programming in GARF, *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, Springer-Verlag.

[22] McAffer, J. (1995) Meta-level Programming with CodA. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*●.

[23] Portillo, A. R., Walker, S., Kirby, G., *et al.* (2003) A Reflective Approach to Providing Flexibility in Application Distribution. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[24] Cazzola, W. and Ancona, M. (2002) mChaRM: a Reflective Middleware for Communication-Based Reflection. *IEEE Distributed System On-Line*, **3**(2).

[25] Object Management Group (1998) *The Common Object Request Broker: Architecture and Specification.*

[26] Kon, F., Román, M., Liu, P., *et al.* (2002) Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York.

[27] Schmidt, D. C. and Cleeland, C. (1999) Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Special Issue on Design Patterns*, **37**(4), 54–63.

[28] Curry, E., Chambers, D., and Lyons, G. (2003) Reflective Channel Hierarchies. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[29] Pietzuch, P. R. and Bacon, J. M. (2002) *Hermes: A Distributed Event-Based Middleware Architecture.*

[30] Cabrera, L. F., Jones, M. B., and Theimer, M. (2001) Herald: Achieving a Global Event Notification Service. *Proceedings of the 8th Workshop on Hot Topics in OS*.

[31] Kiczales, G., Lamping, J., Mendhekar, A., *et al.* (1997) Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*.

[32] Kiczales, G., Hilsdale, E., Hugunin, J., *et al.* (2001) An Overview of AspectJ. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary.

[33] Bergmans, L. and Aksit, M. (2000) Aspects and Crosscutting in Layered Middleware Systems. *Proceedings of the IFIP/ACM (Middleware2000) Workshop on Reflective Middleware*, Palisades, New York.

[34] Chiba., S. (1998) Javassist – A Reflection-based Programming Wizard for Java. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*.

[35] Booch, G. (2001) *Through the Looking Glass*, Software Development.

[36] Tarr, P., Ossher, H., Harrison, W., *et al.* (1999) N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering ICSE'99*.

[37] Ossher, H. and Tarr, P. (2001) Using Multidimensional Separation of Concerns to (re)shape Evolving Software. *Communications of the ACM*, **44**(10), 43–50.

[38] Moreira, R. S., Blair, G. S., and Garrapatoso, E. (2003) Constraining Architectural Reflection for Safely Managing Adaptation. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[39] OMG (2001) *Model Driven Architecture - A Technical Perspective. OMG Document: ormsc/01-07-01*.

[40] Ganek, A. and Corbi, T. (2003) The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, **42**(1).