International Journal of Advanced Science and Technology
Volume 13, December, 2009

Middleware-based Distributed Systems Software Process

[1,2]Liu JingYong, [1]Zhang LiChen, [2]Zhong Yong and [3]Chen Yong

[1]Faculty of Computer Guangdong University of Technology
Guangzhou, 510006, China
[2]Chengdu Institute of Computer Application, Chinese Academy of Sciences
Chengdu, 610041, China
[3]Faculty of Computer science and engineering, Zhongkai University of Agriculture and Engineering Guangzhou, 510225, China
liujycust@sohu.com zhanglichen1962@163.com zhongyong@cimslabsoft.com

Abstract

Middleware facilitates the development of distributed systems by accommodating heterogeneity, hiding distribution details and providing a set of common and domain specific services. It plays a central and essential role for developing distributed systems. However, middleware is considered a mean rather than core elements of development process in the existing distributed systems software process. This paper explains the concept of middleware by categorizes middleware and analysis the problems of current middleware architectures. It also extracts three essential non-functional requirements of middleware and proposes a middleware-based distributed systems software process. The proposed software process consists in five phases: requirements analysis, design, validation, development and testing. The characteristics of middleware are considered in the entire software process. Component-Based Software Engineering, Separation of Concerns, Model-Driven Architecture, formal methods and Aspect Oriented Programming are five active research areas that have been around for several years now. In this paper, we present how these five paradigms can be put together in the context of a new software development method and we show how they can complement each other at different stages in the development life-cycle of middleware-mediated applications.

Keywords: *Distributed systems, Software process, Middleware, Model-Driven Architecture*

1. Introduction

The complexity of distributed systems has increased, which makes a systematic approach to the development of distributed systems very important. To manage the complexity of distributed systems and simplify the development process, middleware is used [1]. A middleware is software that mediates between an application program and a network. It manages the interaction between applications across heterogeneous computing nodes. Using middleware for distributed systems offers several advantages: it eases the distributed system development, increases the portability of the software and improves the system maintenance and reliability.

Research in middleware over the past decade has significantly advanced the quality and feature-richness of general-purpose middleware, such as J2EE, .NET, and CORBA. Middleware serves as the backbone for applications across many domains that have significant societal impact including telecommunications, air traffic control, and industrial automation, among many others. The economic benefits of middleware are significant with up to 50% decrease reported in software development time and costs. Despite these benefits, general-purpose middleware poses numerous challenges when developing real-time systems. First, owing to the stringent demands of real-time systems on quality of service (QoS) (e.g. real-time response in industrial automation) and/or constraints on resources (e.g. memory footprint of embedded medical devices monitoring a patient), the feature-richness and flexibility of general-purpose middleware becomes a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains and/or energy savings. Despite the significant advances in QoS-enabled component middleware, however, applications in important domains (such as large-scale distributed real-time systems) that require simultaneous support for multiple QoS properties are still not well supported. Examples include shipboard combat control systems and supervisory control and data acquisition systems that manage regional power grids. These types of large-scale distributed real-time and embedded (DRE) applications are typified by the following characteristics: stable applications and labile infrastructures. Most DRE systems have a longer life than commercial systems. In the commercial domain, for instance, it is common to find applications that are revised much more frequently than their infrastructure. The opposite is true in many large-scale DRE systems, where the application software must continue to function properly across decades of technology upgrades. As a consequence, it is important that the DRE applications interact with the changing infrastructure through well-managed interfaces that are semantically stable. In particular, if an application runs successfully on one implementation of an interface, it should behave equivalently on another version or implementation of the same interface. End-to-end timeliness and dependability requirements DRE applications have stringent timeliness (i.e., end-to-end predictable time guarantees) and dependability requirements. For example, the timeliness in DRE systems is often expressed as an upper bound in response to external events, as opposed to enterprise systems where it is expressed as events-per-unit time. DRE applications generally express the dependability requirements as a probabilistic assurance that the requirements will be met, as opposed to enterprise systems, which express it as availability of a service. Heterogeneity large-scale DRE applications often run on a wide variety of computing platforms that are interconnected by different types of networking technologies with varying performance properties. The efficiency and predictability of execution of the different infrastructure components on which DRE applications operate varies based on the type of computing platform and interconnection technology.

Second, general-purpose middleware lack out of the box support for modular extensibility of both domain specific and domain-independent features within the middleware without unduly expending extensive manual efforts at retrofitting these capabilities. For example, real-time systems in two different domains as in industrial automation and automotive may require different forms of domain-specific fault tolerance and failover support. Operating conditions for large-scale DRE systems can change dynamically, resulting in the need for appropriate adaptation and resource management strategies for continued successful system operation. In civilian contexts, for instance, power outages underscore the need to detect failures in a timely manner and adapt in real-time to maintain mission-critical power grid operations. In military contexts, likewise, a mission mode change or loss of functionality due to an attack in combat operations requires adaptation and resource reallocation to continue with mission-critical capabilities. Arguably, it is not feasible for general-purpose middleware developers to have accounted for these domain-specific requirements ahead-of time in their design. Doing so would in fact contradict the design goals of middleware, which aim to make them broadly applicable to a wide range of domains, i.e., general purpose.

Building distributed enterprise applications that require the interoperation of multiple components that may be distributed, independently operated, and heterogeneous with respect to language, data model, environment, architecture, and protocols, is a non-trivial task. A middleware is required in order to integrate these diverse software components and to allow them to interoperate effectively. In order to ease the job of software developers and to guide them through the development life-cycle of such enterprise, middleware-mediated systems, new software development method and process need to be proposed. Component-Based Software Engineering (CBSE), Separation of Concerns, Model-Driven Architecture (MDA), formal methods and Aspect Oriented Programming (AOP) are five active research areas that have been around for several years now. In this paper, we present how these five paradigms can be put together in the context of a new software development method and we show how they can complement each other at different stages in the development life-cycle of middleware-mediated applications.

The role of middleware has become more and more important in the distributed systems development process. Properties and functionalities of middleware should be exploited since the early stages of the software process. This drives towards the need of defining a distributed systems software development process based on middleware where both functional and non-functional characteristics of the middleware are taken into account during all phases of the software process.

The rest of this paper is organized as follows. Section 2 is the review of existing works on developing approaches of distributed systems. Section 3 presents detailed discussion about the categories of middleware and the problems of middleware architecture. Section 4 analyzes non-functional requirements of middleware and describes a new middleware based distributed systems software process. In section 5, we present how these five paradigms can be put together in the context of a new software development method and we show how they can complement each other at different stages in the development life-cycle of middleware-mediated applications. A case study is presented in section 6. Finally, section 7 concludes the paper.

2. Related work

Distributed systems development involves many intricate technical details that developers must attend to in order to build robust and efficient applications. To simplify the development process, various technologies have been used to develop distributed systems. Schmidt et al. in [2,3] described how Model-Driven Development (MDD) techniques and tools can be used to specify, analyze, optimize, synthesize, validate, and deploy standards-compliant component middleware platforms that can be customized for the needs of next-generation DRE systems. Results show that coherent integration of MDD tools and component middleware can provide a productive software process for developing DRE systems by modularizing and composing variability concerns and significant challenges remain that must be overcome to apply these technologies to a broader range of DRE systems.

Swapna S. G. et al. in [4] presented project seeks to develop a performance analysis methodology for design-time performance analysis for distributed software systems implemented using middleware patterns and their compositions. The methodology is illustrated on a producer/consumer system implemented using the active object pattern in middleware. And, broader impacts of the methodology for middleware specialization are also described.

Milan J. et al. in [5] deal with middleware services required for the efficient deployment and operation of distributed smart cameras. They focus on services for autonomous and dynamic reconfiguration and develop the services for dynamic reconfiguration using policies. Policies help to specify rules for the reconfiguration process. By evaluation the policy the new task-level configuration of the network is computed. The reconfiguration is implemented using mobile agents in order to achieve a flexible and scalable middleware service.

Amogh K. et al. in [6] described GTQMAP (Graph Transformation for QoS MAPping) model driven engineering tool chain that combines domain specific modeling, to simplify specifying the QoS requirements of DRE systems intuitively, and model transformations, to automate the mapping of domain-specific QoS requirements to middleware-specific QoS configuration options. The automation capabilities of GT-QMAP in the context of three DRE system case studies are evaluated.

3. Middleware

3.1. Categories of middleware

Middleware can be classified into four different categories according to the communication models that are adopted into middleware [7]. They are procedural middleware, message-oriented middleware, transactional middleware, and object middleware.

Procedural middleware supports remote procedure calls (RPC) and communications can be made by using primitives similar to local procedure calls. For example, a server exports several procedures for communication and a client invokes these procedures. Then, procedural middleware marshals an RPC into several messages and vice versa. Procedural middleware does not support scalability, reliability and synchronous invocations very well. However, since its programming interface is easy to understand, it is available on most operating systems.

Message-oriented middleware (MOM) supports the communication between distributed system components by facilitating message exchange. Components can communicate with each other by publishing and subscribing to data using the global data space. Communications are done asynchronously. Message-oriented middleware is particularly well suited for implementing distributed event notification and publish/subscribe-based architectures. However, it only supports at-least once reliability. Thus the same message could be delivered more than once.

Transactional middleware uses the two-phase commit protocol to support distributed transactions. It simplifies the construction of a transactional distributed system. However, it produces an undesired overhead if there is no need to use transactions.

Object middleware is an evolution from procedural middleware. It uses the concept of object-oriented programming to design and implement middleware. This type of middleware enables independent development and distribution of each component since each interaction of components is defined by interfaces. Object middleware integrate most of the capabilities of transactional, message-oriented or procedural middleware. However, the scalability of object middleware is still rather limited [8].

3.2. Problems of current middleware design

Today's distributed computing environment requires a higher degree of modularity for middleware architectures. That high level of modularity is very hard to obtain via traditional architectural methodologies [9]. There are four problems with current middleware architectures.

Lack of capability to match a particular specification with user specific deployment settings easily: there has been a proliferation of middleware specifications to accommodate different requirements that come from many application domains. Vendors must architect the system differently to comply with particular specifications while maintaining platform compatibility and functional consistency. It is very difficult to vendors.

Lack of capability to simultaneously satisfy multiple design requirements: Current middleware architectures lack of methods to address common distribution concerns and particular domain requirements without incurring great architectural complexity and performance penalty.

Lack of adaptability and configurability: in traditional middleware architectures, many systematic properties are not implemented in modules. At the same time, not all of these systematic properties need to participate in middleware operations for every application domain, deployment instance, or runtime condition. Losing of modularity greatly hinders the adaptability and the configurability of middleware platforms.

Lack of performance and memory optimizations: traditional middleware often incur significant throughput and latency overhead. This overhead stems from excessive data copying, internal message buffering strategies that produce non-uniform behavior for different message sizes, and lack of integration with underlying operating system and network QoS mechanisms.

4. Middleware-based distributed systems software process

4.1. Non-functional requirements of middleware

Non-functional requirements address important issues of quality and restrictions for software systems, although some of their particular characteristics make their specification and analysis difficult:

(1)Non-functional requirements can be subjective, since they can be interpreted and evaluated differently by different people;

(2)Non-functional requirements can be relative, since their importance and description may vary depending on the particular domain being considered;

(3)Non-functional requirements can be interacting, since the satisfaction of a particular non-functional requirement can hurt or help the achievement of other non-functional requirement.

A set of ISO/IEC standards are related to software quality, being standards number 9126 [10], 14598-1 and 14598-4 the more relevant ones [11]. The main idea behind these standards is the definition of a quality model and its use as a framework for software evaluation. A quality model is defined by means of general characteristics of software, which are further defined into sub-characteristics in a multilevel hierarchy; at the bottom of the hierarchy appear measurable software attributes. Quality requirements may be defined as restrictions over the quality model. The ISO/IEC 9126 standard fixes 6 top level characteristics: functionality, reliability, usability, efficiency, maintainability and portability. Furthermore, an informative annex of this standard provides an illustrative quality model that refines the characteristics as shown in Fig 1.

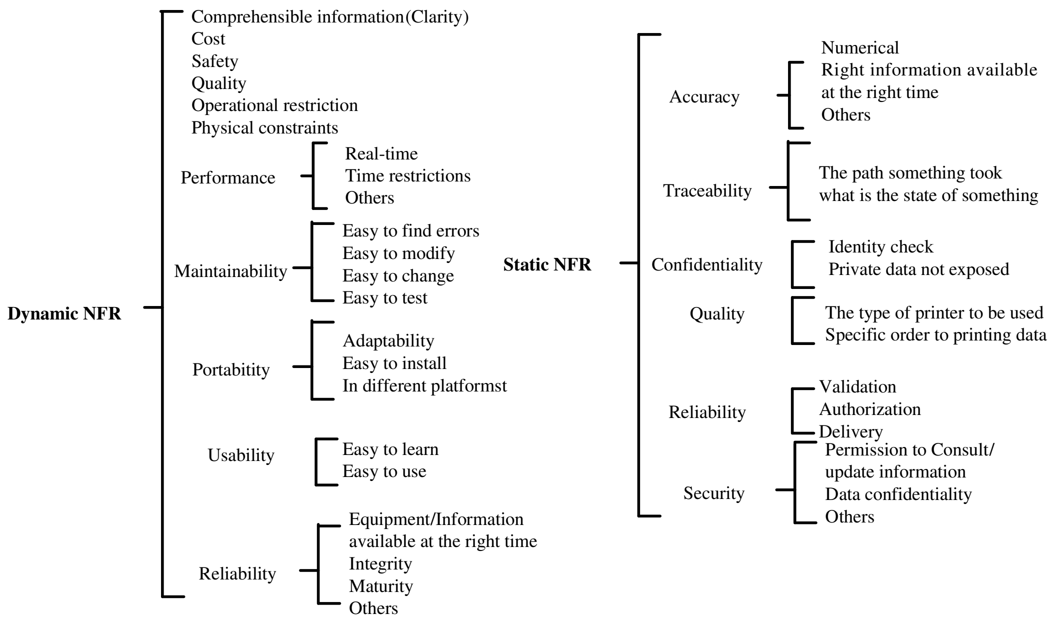


Figure 1. ISO 9126 standard

In order to evaluate these attributes, a metric must be selected and rating levels have to be defined dividing the scale of measurement into ranges corresponding to degrees of satisfaction with respect to the attribute. The rating levels must be defined for each specific evaluation depending on the quality requirements. Finally, a set of assessment criteria combining the measures of attributes are necessary to obtain the rating of the intermediate and top characteristics and, finally, the quality of the product.

According to their relationships with the primary functionality of middleware, Non-functional requirements of middleware can be classified as follows (see Fig. 2). Application aspect, maintenance and composition aspect is based mainly on the study present in [12].

Application aspect can change the internal behavior of middleware. They are additional operational design requirements which a middleware should be configured to support specific target platforms. Examples of such non-functional requirements are memory optimization, error handling, fault tolerance, real-time property, and real-time policy. Since optimizing memory usage is one of the key issues in real-time middleware and it crosscuts the structure of middleware, Memory optimization is viewed as an application aspect of the middleware. Error handling, entangled in the entire middleware, is encapsulated and represented by an error handling aspect. Fault tolerance is another application aspect that influences behavior and structure of a middleware. Additionally, real-time properties and policies are viewed as application aspects as they influence the overall structural behavior of the middleware. Depending on the requirements of a middleware, real-time properties and policies could be further refined. The common characteristic of those aspects is that they extend the primary functionality of middleware.

Maintenance aspect is characteristics that relate to the maintainability of middleware. Examples of such non-functional requirements are logging, tracing and coding rule enforcements aspect. Those non-functional requirements do not carry any operational purposes and they could consume considerable computing resources and major development efforts. The common characteristic of those non-functional requirements is that they are related to the human factors in software engineering.

Composition aspects refer to non-functional requirements that need to be considered when integrating components into middleware. Examples of such non-functional requirements are resource demand, temporal constraints, portability, and flexibility. Each component should have declared resource demands and information of its temporal behavior in its resource demand and temporal constraints respectively. Additionally, the information, such as real-time operating system supported, and other hardware related information, is contained in the portability. Possibilities of extending the component are contained in the flexibility.

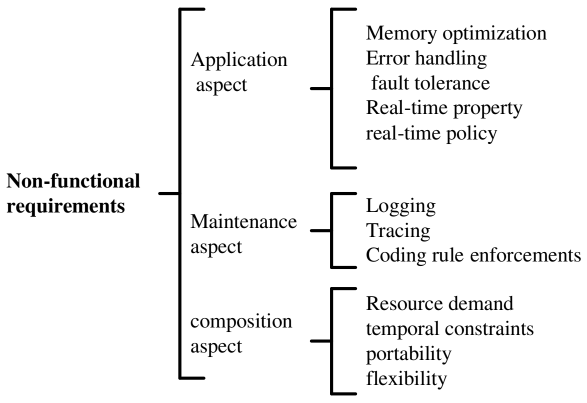


Figure 2. Non-functional requirements classification for middleware

Even if the middleware technologies help in separating business and non-functional requirements, designers and developers face at least two difficulties [13]. Firstly, a middleware consists of a broker plus middleware services such as naming, trading, persistency, transactions, security. The programming of these services is imperative through the concept of API (i.e., offered interfaces). Thus, the code using these services is interleaved with the business code. Secondly, designers and developers often cannot afford the mastering and even, the study of the internals of these services when they want to combine them. Therefore, the objective of the reuse as black boxes of middleware services by third parties is not attained by middleware technologies. With software component frameworks, the non-functional code is automatically generated or generic, and not interleaved with the business code. We claim that the container is the right place for non-functional services' providers to put the code using their services. The second idea developed is that during design, the container is also the right place for modeling the combination of non-functional services.

One of the primary goals of the container is the transparency of the non-functional requirements to the software component. The non-functional requirements must be provided to a software component without any modification of its core logic if possible. Nevertheless, applications must sometimes be aware of the non-functional services they use. At the architectural level, the container helps the designer in the separation of the business and technical parts with the concept of view. During execution, the container is responsible for managing and combining accesses to external non-functional services. The container realizes the combination of non-functional services, except the brokering service which is often provided by the middleware. Connectors realize also non-functional requirements, but mainly the ones concerning the interactions between software components -- e.g., the brokering service. Containers may also be responsible for the portability onto different middleware technologies. The container provider prepares artifacts to simplify the modeling of the usage and the adaptation to the non-functional services. The first task is to combine the non-functional services using patterns he/she identified during his/her expertise. The new artifacts are UML architectural models of the container and a UML framework for using the non-functional services in UML models. The UML framework can be a profile for using the non-functional services; it uses UML extension mechanisms such as stereotypes, tagged values, and constraints. The second task is to help the application developer in transparently integrating the non-functional requirements to implementation components. This can be done with code and configuration descriptors generators that automatically construct stubs, skeletons, XML descriptors files. If the application is developed with object-oriented languages, the code generators make the most of object mechanisms such as inheritance, delegation, and meta-object protocol.

Figure 3 shows the UML extensions for non-functional requirements (NFR) modeling. The NFR Modeling package (stereotyped as profile) defines how the elements of the domain model extend meta-classes of the UML meta-model. A UML framework can provide a profile to architects and designers for modeling non-functional dependencies. One or more non-functional requirements can be attached to the operations (i.e., pre-/post-conditions) and other constraints in OCL, the attributes, and other model elements of a software component [14, 15]. Non-functional requirements can also be attached to the links between the software components in order to configure the middleware for instance. The main UML extension mechanisms are constraints, tagged values, and stereotypes. These extensions are used for documentation purposes and for directing code and configuration descriptors generation. Attaching constraints and tagged values is the simplest way to add non-functional requirements to a model element. A constraint consists in specifying more semantics as an expression in a designated constraint language. Constraints are gaining more and more importance in UML. A tagged value consists of a name and its associated value. By definition, constraints and tagged values are simple and very extensive-since there are very few limitations on their usage. The resulting contract between the service and the software component can be too fine-grained and spread over multiple model elements. Therefore, they may be complex to use for configuring non-functional services.

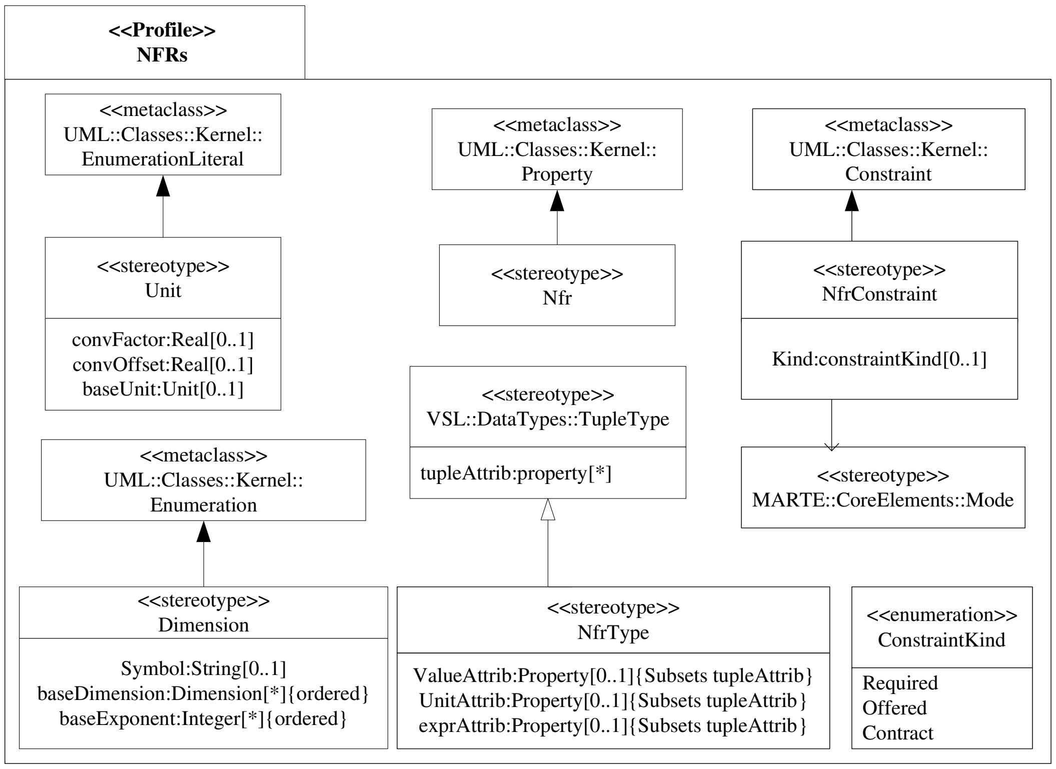


Figure 3. UML profile diagram for NFRs modeling

4.2. The distributed systems software process

The distributed systems software development process based on middleware is divided in five phases. Fig. 4 depicts the whole software development process.

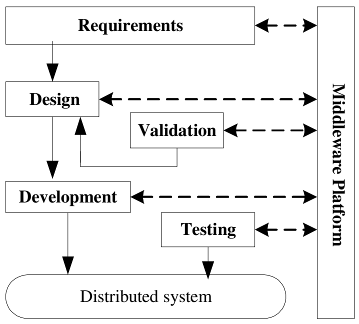


Figure 4. Distributed systems software process based on middleware.

The first phase is a profound analysis of the requirements. This phase is dedicated to identifying functional requirements and non-functional requirements of the distributed system and middleware. To elicit functional requirements, use case diagrams and templates are used. To elicit non-functional requirements, check-lists, lexicons, and conflict resolution rules are used [16]. Based on the results coming from the requirements phase, a middleware platform that properly addresses the application domain concerns is selected.

Middleware specifications are not trivial to be understood, as the middleware itself is usually very complex [17]. Firstly, middleware systems have to hide the complexity of underlying network mechanisms from the application. Secondly, the number of services provided by the middleware is increasing, e.g., the CORBA specification contains fourteen services. Finally, in addition to hide communication mechanisms, the middleware also have to hide fails, mobility, changes in the network traffic conditions and so on. On the point of view of application developers, they very often do not know how the middleware really works. On the point of view of middleware developers, the complexity places many challenges that include how to integrate services in a single product or how to satisfy new requirements of emerging applications [18]. Formal description techniques have been used together middleware in the Reference Model - Open Distributed Processing [19], in which the trader service is formally specified in LOTOS. The Z notation and high level Petri nets also have been adopted for specifying CORBA services, the naming service, the event service and the security service [20]. All those works, however, do not adopt software architecture principles for structuring the service descriptions. In terms of software architecture, a few Architecture Description Languages include the possibility of describing behavior. However, there are not tools that enable us to check behavior properties. Medvidovic [21] has observed the convergence of middleware and software architecture principles. However, he does adopt a formal approach. Finally, it is possible to note that the software architecture principles are widely adopted to build distributed applications, but its benefits are rarely applied to middleware that connect them. For example, the use of LOTOS allows the checking of particular behavioral properties of middleware systems, e.g., deadlock, execution sequences. Additionally, the language allows automatically generating test and checks the behavioral equivalence (e.g., strong equivalence, branching equivalence, weak equivalence) between different middleware models and different middleware service compositions. For example, if one desires to replace a message oriented middleware by a procedural middleware, it is possible to check if their behaviors are equivalent. Finally, a formal specification eliminates ambiguities in the middleware specification and provides a better understanding of what is actually described [22].

In the design phase, the distributed system will be designed considering both the requirements and the constraints posed by the system and middleware. Formal methods and UML models can be used in this phase [23].

The validation phase is in charge of validating the application design against both functional and non-functional requirements. Also in this phase, the middleware characteristics have to be considered since they can affect the application validation. In the development phase, all kinds of programming techniques can be used. The source code of classes and aspect is generated. And the final software system is built on top of the middleware platform.

Finally, in the testing phase, the distributed system and middleware platform is tested. In this phase, Object-oriented testing approaches can be used.

5. Applying AOP and MDA to middleware-based distributed systems software process

5.1. Aspect-oriented programming

Object-Oriented Programming (OOP) has been the dominant programming methodology that is being used in all kinds of software development today. The main focus of OOP is to find a modular solution for a problem by breaking down the system into a collection of classes that encapsulates state and behavior. However, In Object-Oriented Programming, crosscutting concerns are elements of software that can not be cleanly captured in a method or class. Accordingly, crosscutting concerns has to be scattered across many classes and methods. OOP fails to provide a robust and extensible solution to handle these crosscutting concerns. AOP [24] is a new modularity technique that aims to cleanly separate the implementation of crosscutting concerns. It builds on Object-Orientation, and addresses some of the points that are not addressed by OO. AOP provides mechanisms for decomposing a problem into functional components and aspectual components called aspects. An aspect is a modular unit of crosscutting the functional components, which is designed to encapsulate state and behavior that affect multiple classes into reusable modules. Distribution, logging, fault tolerance, real-time and synchronization are examples of aspects. The AOP approach proposes a solution to the crosscutting concerns problem by encapsulating these into an aspect, and uses the weaving mechanism to combine them with the main components of the software system and produces the final system. We think that the phenomenon of handling multiple orthogonal design requirements is in the category of crosscutting concerns, which are well addressed by aspect oriented techniques. Hence, we believe that middleware architecture is one of the ideal places where we can apply aspect oriented programming (AOP) methods to obtain a modularity level that is unattainable via traditional programming techniques. To follow that theoretical conjecture, it is necessary to identify and to analyze these crosscutting phenomena in existing middleware implementations. Furthermore, by using aspect oriented languages, we should be able to resolve the concern crosscutting and to yield a middleware architecture that is more logically coherent. It is then possible to quantify and to closely approximate the benefit of applying AOP to the middleware architecture.

5.2. Model-driven architecture

MDA [25] is a framework proposed by OMG for the software development, driven by models at different abstraction levels. MDA relies on the separation of the business logic of a system from its implementation. To achieve this, MDA defines two types of models: the Platform-Independent Model (PIM) and the Platform-Specific Model (PSM). PIM captures system behavior and functionality, while PSM captures information about details of system implementation. The MDA development process primarily involves three steps. First the PIM is developed. The objective is to capture the high level functional requirements of the application. In the second step, transformation rules are used to transform the PIM into one or more Platform-Specific Models. A PSM is customized to describe the system in terms of the particular implementation platform. The third step involves the conversion of the PSM into application code. Typically, steps two and three are automated by the use of automated tools. It is the first step in the process that involves creativity and manual work. In general, MDA is a useful approach towards reasoning about the impact of middleware on the behavior of software systems.

MDD attempts to raise the level of abstraction by which software and systems engineers carry out their tasks. This is done by emphasis the use of models (i.e., abstractions) of the artifacts that are developed during the engineering process. Models are representations of phenomena of interest, and in general are usually easier to modify, update, and manipulate than the artifact or artifacts that are being represented. Models are expressed using a suitable modeling language; UML is a widely used standard in MDD. MDD is not a development method or process; it can be implemented in a number of ways, e.g., via Extreme Programming, the Rational Unified Process, the B-Method, or a refinement calculus. The key element in MDD is the construction and transformation of models that are fit for the purposes of the development project. The languages and processes used in construction and transformation will vary from project to project.

The MDA guide is vague in its definition of MDA and the notion of refinement. The guide defines MDA in terms of PIM, PSM, and additional models such as domain models. Refinement is defined informally as a process of transforming MDA models (e.g., PIM to PSM, PSM to code, PIM to PIM). The MDA guide distinguishes PIM and PSM as models at different levels of abstraction, e.g., a PIM is at a higher level of abstraction than a PSM. However, years of research on refinement calculi and programming methodology, particularly on wide-spectrum languages, suggest that distinctions such as this are not helpful: it is more productive to think in terms of specifications that have different properties. For example, in predicative programming, programs are a special kind of specification. They are implementable and immediately executable on a machine. Similarly, in refinement calculus, specifications are a special kind of program. they are not always executable, but one can test for feasibility, and they are written in a unified language. To formally define refinement in MDA, there are four alternatives. One could translate the core languages used in MDA i.e., UML, or a subset of UML into a formal language such as Z, B, LOTOS or specification statements. Work has been carried out on expressing such translations, but it all suffers from limitations, e.g., incompleteness, difficulties in achieving consistency, etc. A second alternative is to promote a formal definition of refinement e.g., weakest preconditions and express it in MDA terms, e.g., in UML. It is debatable whether UML is well suited to expressing formal definitions of refinement.

The specification of the Object Constraint Language (OCL) [26] is a part of the UML specification, and it is not intended to replace existing formal languages, but to supplement the need to describe the additional constraints about the objects that cannot be easily represented in graphical diagrams, like the interactions between the components and the constraints between the components' communication [27]. In object-oriented modeling, a graphical model, such as a class diagram, is not enough for a precise unambiguous specification. OCL is designed to solve this problem. It facilitates the specification of model properties in a formal yet comprehensive way. By combining the power of the straightforward, graphical UML modeling and the textual, accurate OCL constraints, these kinds of information can be specified in this formal way.

OCL has the characteristics of an expression language, a modeling language and a formal language. An OCL expression is guaranteed to be without side effects since it is an expression language, and thus cannot change anything in the model, although an OCL expression can be used to specify the state changes of the system. OCL is not a programming language, but a modeling language. So it is impossible to write program logic or flow-control in OCL. All implementation issues are likewise out of the scope of OCL. OCL is also a formal language where all constructs have a formally defined meaning; in other words, it is unambiguous. Furthermore, OCL is strongly typed.

The main idea behind OCL is "Design By Contract" [28]. By applying this, the responsibility of the parties is made unambiguous and can be formally described. An OCL constraint consists of the precondition, the post-condition and the invariant. The contract is a way of establishing who does what by stating, first, what must be true for the caller (client) to request a service from the callee (server) (precondition), and, what must be true when the callee finishes providing the service (post-condition). The invariant must be true when a routine is called and when it terminates, but not necessarily when it is executing. By the principle of "Design By Contract", and specifying these three constraints, the services provided by the server are exposed, but not the details of the implementation of the services.

On the other hand, the callee will know when exactly a service can be provided (available), and the caller will know when exactly it can request the service. In case of exceptions, it is easy to find out who caused the exception: if the precondition is false, the caller broke the contract; if the post-condition is false, the callee broke the contract; if the invariant is false, the callee class broke the contract.

Since OCL is a textual extension of the graphical UML modeling language, an OCL specification is always unambiguous and precise. It also provides better documentation to the visual models. It can be used during the modeling and specification. Since OCL is an expression language, it can be checked without an executable system. All these features turn out to be useful in representing QoS properties, which can be represented by the combination of precondition, post-condition and invariant in OCL. The QoS attributes are represented by the member variables of the class, and the QoS actions are represented by the methods. They are checked at run time, before and after the calls so that the change of the QoS parameters of the system is monitored in a timely basis.

The precondition has to be satisfied before the method can be called, and the post-condition has to be satisfied at the time the method returns. It is easy to find out which step causes exceptions if any. The methods are called in a loop-like fashion, so, whenever a change of the QoS parameter is observed, the corresponding methods are called and the changes are made accordingly and the necessary notification is done at the same time. The QoS specification is integrated in the overall system design in this fashion. In this way, the satisfaction of the QoS requirements is guaranteed and the change of the QoS properties is under observation and control, as well.

Although QoS properties and associated metrics have been widely used in networking, there is no standard vocabulary for discussing the QoS as it relates to the distributed computing and component-based solutions [29], especially when the QoS properties are applied on variant platforms and when the different aspects of the QoS interact with each other. A standard vocabulary is the first step toward progressing Model Driven Architecture that includes QoS parameterization and/or QoS contracts. MDA provides an open, vendor-neutral environment for the integration of different distributed application software. MDA aims to separate the business or application logic from the underlying platform technology. Its standards are made up of the UML, Meta-Object Facility (MOF), XML Meta-Data Interchange (XMI), and Common Warehouse Meta-model. Platform-independent applications built using MDA and the associated standards can be realized on a range of platforms.

The MDA design initiative assists during the interaction between the different platforms and different middleware. Middleware environments started out providing the interoperability using the architectures that are standard, proprietary, or somewhere in the middle. Progressively, more and more services and more powerful middleware have been added to the overall architecture, thus, it is more difficult to ensure the interoperability of these middleware. To efficiently solve this problem, MDA is designed by applying the component and modeling technology and putting the whole picture together.

5.3. Applying AOP and MDA to the software process

The distributed systems software development process based on aspect-oriented middleware is divided in five phases. Fig. 5 depicts the whole software development process.

The first phase is a profound analysis of the requirements. The phase includes three steps:

Step one handles the non-functional requirements and then identifies which of those are crosscutting.

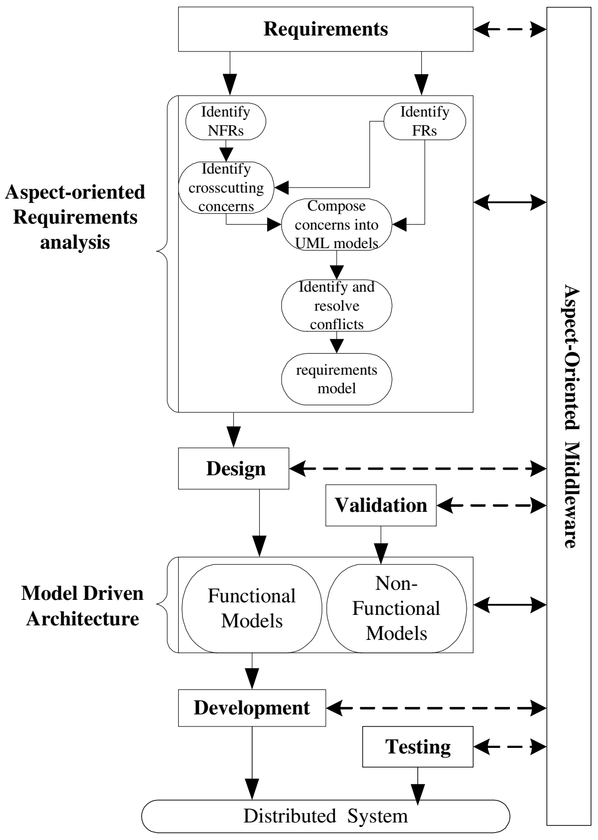

Requirements
Aspect-oriented Requirements analysis
Identify NFRs
Identify FRs
Identify crosscutting concerns
Compose concerns into UML models
Identify and resolve conflicts
requirements model
Design
Validation
Model Driven Architecture
Functional Models
Non-Functional Models
Development
Testing
Distributed System
Aspect-Oriented Middleware

Figure 5. Distributed systems software process based on aspect-oriented middleware

Step two performs a traditional specification of functional requirements, in this case, using an UML-like approach where the use case model is the main specification technique.

Step three starts by composing functional requirements with aspects; then it identifies and resolves conflicts that may arise from the composition process.

The concepts of overlapping, overriding and wrapping [30] can be adopted to define the composition part of the model. Overlapping indicates the requirements of the aspect modifies the functional requirements they transverse. In this case, the aspect requirements may be required before the functional ones, or they may be required after them. Overriding indicates the requirements of the aspect superpose the functional requirements they transverse. In this case, the behavior described by the aspect requirements substitutes the functional requirements behavior. Wrapping indicates the requirements of the aspect encapsulate the functional requirements they transverse. In this case, the behavior described by the functional requirements is wrapped by the behavior described by the aspect requirements.

In the design phase, the distributed system will be designed considering both the requirements and the constraints posed by the system and middleware. Using the MDA approach to produce the platform specific models includes five steps [31] (see Fig. 6):

Step one: Create the PIM for the distributed system.

Step two: Select the target middleware and create the generic middleware aspects.

Step three: Transform PIM to enhanced PIM using the application converter.

Step four: Transform the generic aspects to enhanced aspects using the aspect converter.

Step five: Weave the enhanced aspects into the enhanced PIM to produce the PSM.

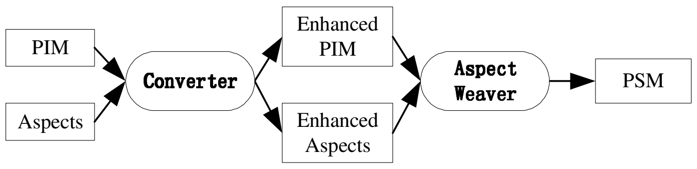


Figure 6. Process view of the PSM generation

The validation phase is in charge of validating the application design against both functional and non-functional models. Also in this phase, the middleware characteristics have to be considered since they can affect the application validation. Model-based analysis techniques can be used for validation purposes [32]. Because it provides a way for the design-time analysis of distributed systems enabling rapid evaluation of design alternatives with respect to given performance measures before committing to a specific platform.

In the development phase, the source code of classes and aspect is generated. And the distributed system is built on top of the aspect-oriented middleware platform.

Since aspect may affect the behavior of one or more classes through advice and introduction. Traditional testing techniques, such as unit testing, integration testing, are not applicable to test aspect in the testing phase. Some aspect-oriented testing approaches [33], such as data-flow-based unit testing, state-based testing approach, and model-based testing approach can be used in this phase.

To address the middleware development, principled methods are needed to specify, develop, compose, integrate, and validate the application and middleware software used by DRE systems. These methods must enforce the physical constraints of DRE systems, as well as satisfy the system’s stringent functional and non-functional requirements. Achieving these goals requires a set of integrated Model Driven Middleware (MDM) tools that allow developers to specify application and middleware requirements at higher levels of abstraction than that provided by low-level mechanisms, such as conventional general-purpose programming languages, operating systems, and middleware platforms. Different functional and systemic properties of DRE systems via separate middleware and platform-independent models are applied. Domain-specific aspect model weavers can integrate these different modeling aspects into composite models that can be further refined by incorporating middleware and platform-specific properties. Different but interdependent characteristics and requirements of DRE system behavior (such as scalability, predictability, safety, and security) are specified via models. Model interpreters translate the information specified by models into the input format expected by model checking and analysis tools. These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints. Tool-specific model analyzers can also analyze the models and predict expected end-to-end QoS of the constrained models. Platform-specific code and metadata that is customized for a particular QoS-enabled component middleware and DRE application properties, such as end-to-end timing deadlines, recovery strategies to handle various run-time failures in real-time, and authentication and authorization strategies are modeled at a higher level of abstraction. Middleware and applications by assembling and deploying the selected components end-to-end using the configuration metadata are synthesized by MDM tools. In the case of legacy components that were developed without consideration of QoS, the provisioning process may involve invasive changes to existing components to provide the hooks that will adapt to the metadata. The changes can be implemented in a relatively unobtrusive manner using program transformation systems.

6. Case Study: CORBA Based applications

CBSE is one of the paradigms of distributed system development that is most popular at present. Good proof of it is the expansion that it has in platforms such as EJB, COM or CORBA Component Model (CCM). However, the description of the dependencies of the components and their subsequent implementation causes the appearance of crosscutting. This situation makes difficult not only the adaptability of developed components when new requirements appear but also their reusability in domains different from those for which they were designed. AOP is able to contribute to CBSE providing the necessary mechanisms to remove the dependencies from components when designing and implementing them, eliminating so the cause of crosscutting and promoting a high degree of flexibility, adaptability, and reusability which are essentials in Component-Based Systems (CBS). UML is used as modeling language due to it being a standard. Dependencies among components are identified and treated successfully to avoid the appearance of crosscutting. This also favors the description of components fully independent of the problem domain.

For example, in case of CORBA the platform is specified by a set of interfaces and usage patterns that constitute the CORBA Core Specification. The CORBA platform is independent of operating systems and programming languages. The OMG Trading Object Service specification (consisting of interface specifications in OMG Interface Definition Language (IDL)) can be considered to be a PIM from the viewpoint of CORBA, because it is independent of operating systems and programming languages. When the IDL to C++ Language Mapping specification is applied to the Trading Service PIM, the C++-specific result can be considered to be a PSM for the Trading Service, where the platform is the C++ language and the C++ ORB implementation. Thus the IDL to C++ Language Mapping specification (IDLC++) determines the mapping from the Trading Service PIM to the Trading Service PSM.

The UML Profile for EDOC specification is another example of the application of various aspects of MDA. The extension of UML concepts allows us to model a component, its interfaces, events and QoS contracts in a graphical manner. Weis, T., et al. in [34] demonstrated how UML can be extended to support contracts for non-functional aspects in general and QoS in special. The UML drawings can help a lot to gain an overview of all components, interfaces and their possible QoS contracts during the application design phase. It defines a set of modeling constructs that are independent of middleware platforms such as EJB, CCM, etc. A PIM based on the EDOC profile uses the middleware - independent constructs defined by the profile and thus is middleware -independent. In addition, the specification defines formal meta-models for some specific middleware platforms such as EJB, supplementing the already existing OMG meta-model of CCM. The specification also defines mappings from the EDOC profile to the middleware meta-models. For example, it defines a mapping from the EDOC profile to EJB. The mapping specifications facilitate the transformation of any EDOC-based PIM into a corresponding PSM for any of the specific platforms for which a mapping is specified [35].

In the following specification, formal technique LOTOS is applied to define the CORBA. In this case, the CORBA receives a request from the server and sends it to client. After being processed, the reply is sent back to the client via the middleware. At this abstraction level, the software architecture does not present details on how this task is actually performed. The behavior of the connector CORBA is specified as the temporal ordering of events executed in the CORBA interface. The CORBA interface is made up of several other interfaces such as dynamic invocation, stub, ORB, static skeleton, dynamic skeleton and POA interfaces. The operations defined in each of the aforementioned interfaces are passed to the middleware through the event "invClt ? s : Service ? op : OPER;", where s is the name of service being request and op the operation.

```
process CORBA [invClt, terClt, invSrv, terSrv] : noexit :=
invClt ? s : Service ? op : OPER;
invSrv ! s ! op;
terSrv ! s ? r : RESULT;
terClt ! s ! r;
CORBA [invClt, terClt, invSrv, terSrv]
endproc
The ORB (connector) as shown in the following:
process CORBA [invClt, terClt, invSrv, terSrv] : noexit :=
hide inv, ter in
(( Naming [inv, ter] ||| Event [inv, ter] ||| Persistent [inv, ter] |||
LifeCycle [inv, ter] ||| Concurrency [inv, ter] ||| Externalization [inv, ter] |||
Relationship [inv, ter] ||| Transaction [inv, ter] ||| Query [inv, ter] |||
Licensing [inv, ter] ||| Property [inv, ter] ||| Time [inv, ter] |||
Security [inv, ter] ||| Trading [inv, ter] )
||
ServiceOrdering [inv, ter] )
|[inv, ter]|
ORB [inv, ter, invClt, terClt, invSrv, terSrv] (0)
where
...
Endspec
```

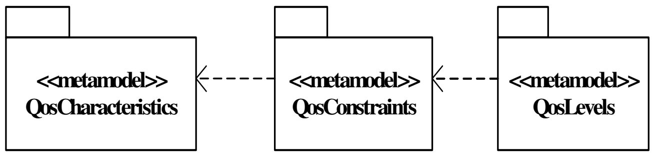

<<metamodel>> QosCharacteristics
<<metamodel>> QosConstraints
<<metamodel>> QosLevels

Figure 7. meta-models in the QoS meta-model

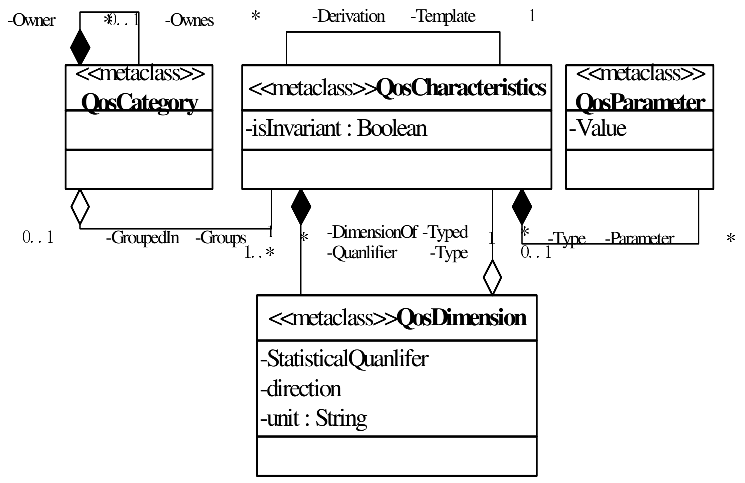


Figure 8. QoSCharacteristic diagram

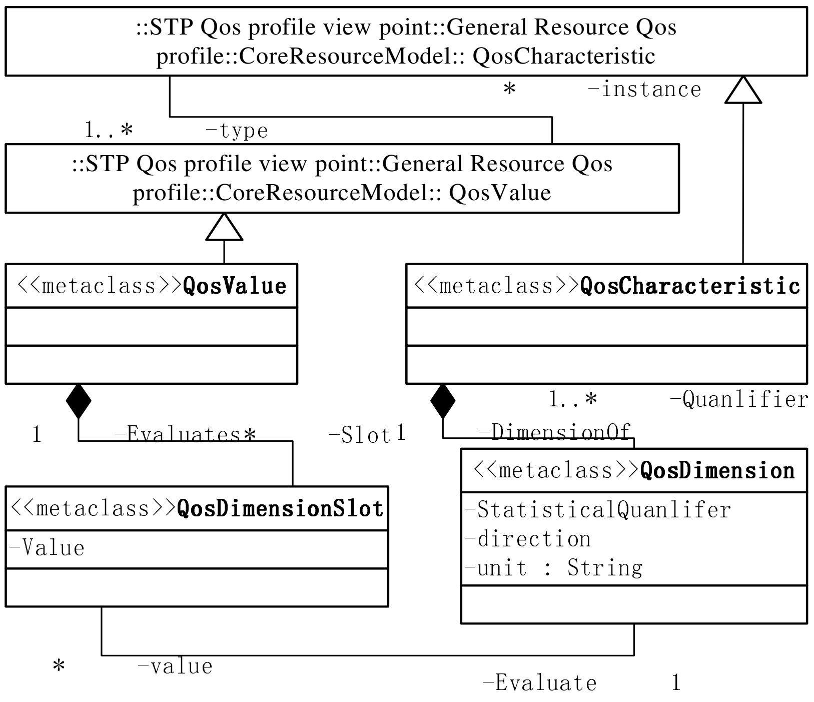

::STP Qos profile view point::General Resource Qos profile::CoreResourceModel:: QosCharacteristic
*
-instance
1..*
-type
::STP Qos profile view point::General Resource Qos profile::CoreResourceModel:: QosValue
<<metaclass>>QosValue
<<metaclass>>QosCharacteristic
1..*
-Quanlifier
1
-Evaluates
*
-Slot
1
-DimensionOf
<<metaclass>>QosDimension
-StatisticalQuanlifer
-direction
-unit : String
<<metaclass>>QosDimensionSlot
-Value
*
-value
-Evaluate
1

Figure 9. QoSValues diagram

Although the CCM is a major step forward to support the design and implementation of component based distributed telecommunication systems, it has not sufficiently addressed the non-functional, i.e. QoS related aspects of distributed systems. Instead of just adding a fixed set of QoS categories to the modeling approach we decided to design a generic, flexible and multi-category architecture. For that reason we define a meta-model that allows a generic specification of QoS contracts, which can be negotiated between software components. We have integrated this meta-model with the CCM meta-model and the UML meta-model. Both meta-models are widely used to model the functional aspects of distributed systems. Due to our meta-model approach it is possible to integrate the same QoS meta-model with different approaches for the functional design. The extension of UML concepts allows us to model a component, its interfaces, events and QoS contracts in a graphical manner. The UML drawings can help a lot to gain an overview of all components, interfaces and their possible QoS contracts during the application design phase. QoS is defined by UML as shown in Figure 7, Figure 8, Figure 9 and Figure 10, and figure 11 shows aspect model of QoS framework.

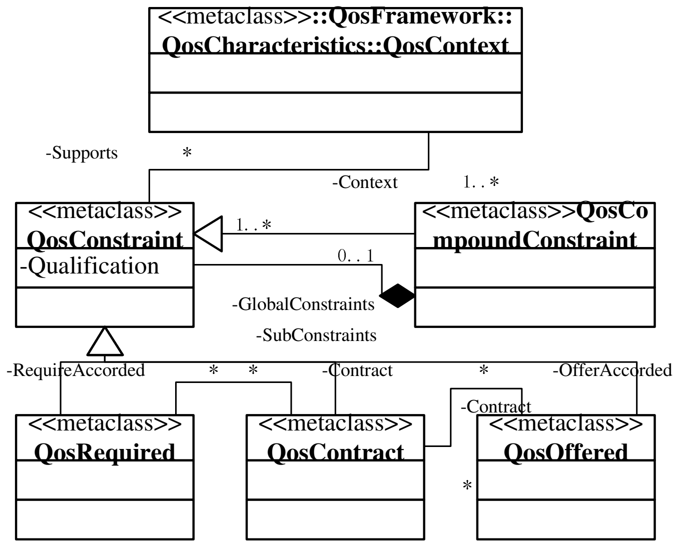

<<metaclass>>::QosFramework::QosCharacteristics::QosContext
-Supports
*
-Context
1..*
<<metaclass>> QosConstraint
-Qualification
1..*
0..1
<<metaclass>>QosCompoundConstraint
-GlobalConstraints
-SubConstraints
-RequireAccorded
*
*
-Contract
*
-OfferAccorded
-Contract
<<metaclass>> QosRequired
<<metaclass>> QosContract
<<metaclass>> QosOffered
*

Figure 10. QoSConstraint diagram

7. Conclusion

Networked heterogeneous systems pose a great challenge for distributed systems. Middleware provides a critical link between the vast resources and the application domain that simplifies development and provides robust and reliable access to resources, helps optimize resource utilization, and facilitates the generation of stable distributed software. However, middleware is considered a mean rather than core elements of development process in the existing distributed systems software process. The proposed software process consists in five phases: requirements analysis, design, validation, development and testing. The characteristics of middleware are considered in the entire software process. Component-Based Software Engineering, Separation of Concerns, Model-Driven Architecture, formal methods and Aspect Oriented Programming are five active research areas that have been around for several years now. In this paper, we present how these five paradigms can be put together in the context of a new software development method and we show how they can complement each other at different stages in the development life-cycle of middleware-mediated applications. The further work is devoted to developing tools to support the automatic generation of model and code.

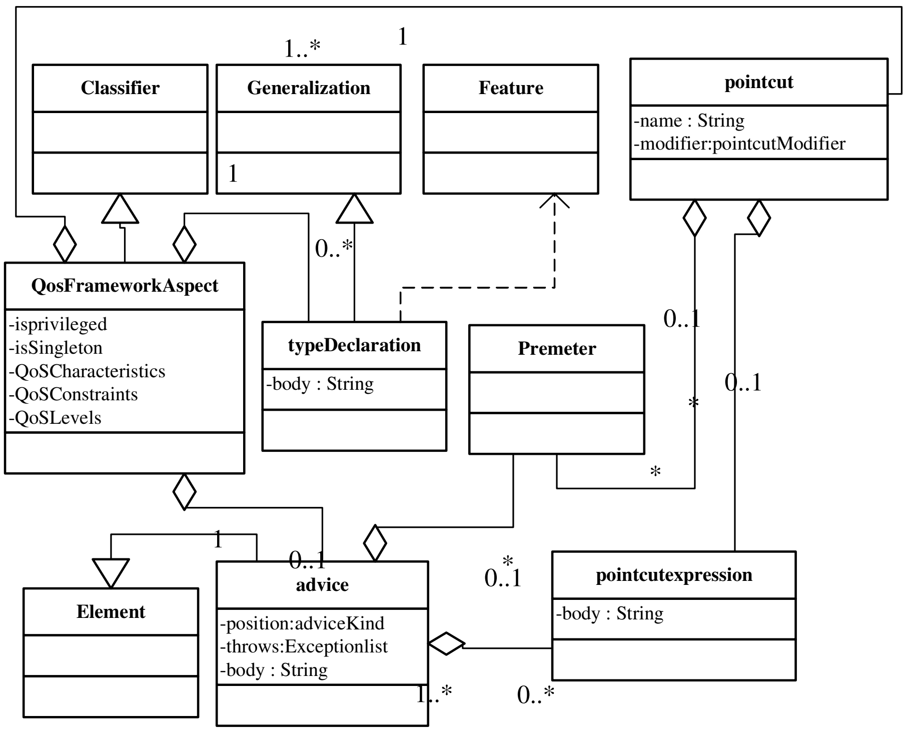


Figure 11. Aspect model of QoS framework

Acknowledgments

This work is supported by the Major Program of National Natural Science Foundation of China under Grant No.90818008, and by the National Natural Science Foundation of China under Grant No.60774095, and by the Natural Science Foundation of Guangdong Province of China under Grant No.07001774.

References

[1]N. Loughran, N. Parlavantzas, and M. Pinto, “Survey of Aspect-Oriented Middleware”, AOSD-Europe Project Deliverable, No: AOSD-Europe-ULANC-10. Editor(s): N. Loughran, M.Pinto, June, 2005.

[2]Gan Deng, Douglas C. S.,and Aniruddha Gokhale, et al., “Modularizing Variability and Scalability Concerns in Distributed Real-time and Embedded Systems with Modeling Tools and Component Middleware”, Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, Gyeongju, Korea, April 24-26, 2006, pp.327-334.

[3]A. Gokhale, K. Balasubramanian, and J. Balasubramanian, et al., "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications", The Journal of Science of

Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture, Vol. 73, Issue 1, September 2008, pp.39-58.

[4]Gokhale, S.S., Vandal, P., and Gokhale, A.,et al., "Model-Driven Performance Analysis Methodology for Distributed Software Systems", Proceedings of the Parallel and Distributed Processing Symposium, Long Beach,CA, March 26-30,2007, IEEE Computer Society, pp.1-10.

[5]Jovanovic, M. Rinner, B., "Middleware for Dynamic Reconfiguration in Distributed Camera Systems",Fifth Workshop on Intelligent Solutions in Embedded Systems, Leganes, Madrid,ES, June 21-22,2007, IEEE Computer Society, pp.139-150.

[6]Amogh Kavimandan, Aniruddha Gokhale,"Automated Middleware QoS Configuration Techniques for Distributed Real-time and Embedded Systems", IEEE Real-Time and Embedded Technology and Applications Symposium, St. Louis, MO, United States, April 22-24,2008, IEEE Computer Society, pp.93-102.

[7]Jiyong Park,Saehwa Kim and Wooseok Yoo, et al.,"Designing Real-Time and Fault-Tolerant Middleware for Automotive Software", SICE-ICASE 2006 International Joint Conference, Busan, Korea, October 18-21, 2006, IEEE Computer Society, pp.4409-4413.

[8]W. Emmerich, "Software Engineering and Middleware: A Roadmap", Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000, ACM Press, pp.76-90.

[9]C.Zhang, H-A.Jacobsen, "Quantifying Aspects in Middleware Platforms ",Proceedings 2nd International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, March 17-21, 2000, ACM Press, pp.130-139.

[10]International Organization for Standarization, ISO/IEC Standard 9126: Software Engineering – Product Quality. 2004.

[11]International Organization for Standarization,ISO/IEC14598 Information Technology - Software product evaluation. 2001.

[12]T. Aleksandra ,N. Dag ,and H. Jorgen, et al., " Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Sofftware ",Journal of Embedded Computing, IOS Press, vol.1, Jan. 2005, pp.17-37.

[13]D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel,"Integration of non-functional properties in containers",Proceedings of the 6th International Workshop on Component-Oriented Programming, 2001,pp.1-8.

[14]Liming Zhu,Gorton, I. ,"UML Profiles for Design Decisions and Non-Functional Requirements",Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, 20-26 May 2007 pp.8-9.

[15]Clemente, P. J., Sánchez, F., and Perez, M. A. "Modelling with UML Component-based and Aspect Oriented Programming Systems",Seventh International Workshop on Component-Oriented Programming at European Conference on Object Oriented Programming (ECOOP). Málaga, Spain.,2002,pp.1-7.

[16]Wehrmeister, M.A., Freitas, E.P., and Pereira, C.E., et al., "An Aspect-Oriented Approach for Dealing with Non-Functional Requirements in a Model-Driven Development of Distributed Embedded Real-Time Systems ", 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, Santorini Island, Greece, May7-9, 2007, IEEE Computer Society, pp.428-432.

[17]Campbell, Andrew T., Coulson, Geoff and Kounavis, Michael E, “Managing Complexity: Middleware Explained”, IT Professional, IEEE Computer Society, Vol 1(5), 1999,pp. 22-28.

[18]Venkatasubramanian, Nalini, “Safe Composability of Middleware Services”,Communications of the ACM, Vol 45(6), June,2002,pp. 49-52.

[19]International Organization for Standarization, ISO 10476-1 “Reference Model of Open Distributed Processing (Part I) -Overview”, July,1995.

[20]Basin, David, Rittinger, Frank and Viganò, Luca “A Formal Analysis of the CORBA Security Service”, Lecture Notes in Computer Science, 2002, No. 2272,pp.330-349.

[21]Medvidovic, Nenad, "On the Role of Middleware in Architecture-based Software Development",14th International Conference on Software Engineering and Knowledge Engineering 2002, pp. 299-306.

[22]Rosa, N. S.; Cunha, Paulo R F ., "Adopting LOTOS and Software Architecture Principles for Formalising Middleware Behaviour",In: SBRC 2004, Gramado. XXII Simpósio Brasileiro de Redes de Computadores, 2004. v. 1. pp. 525-538.

[23]L. Lavazza, G. Quaroni,M. Venturelli, "Combining UML and formal notations for modeling real-time systems", ACM SIGSOFT Software Engineering Notes, vol. 26, Sep. 2001, pp.196-206.

[24]Kiczales, G., et al., “Aspect-Oriented Programming”, Proc. of ECOOP, Lecture Notes in Computer Science 1241, Springer-Verlag, 1997, pp.220-240.

[25]S. Mellor, K. Scott, A. Uhl, and et al., MDA Distilled, Addison-Wesley Professional, 2004.

[26] J. Warmer, A. Kleppe,"The Object Constraint Language", Addison-Wesley, 1999.

[27]Yang, Chunmin , Bryant, Barrett R. , and Burt, Carol C. , et.al ,"Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing",Journal of Integrated Design & Process Science,Vol 8 , Issue 2,2004,pp. 137 - 149.

[28] Frankel, D. S., "Model Driven Architecture:Applying MDA to Enterprise Computing", OMG Press.

[29] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A., and Auguston, M., "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models". Proceedings of EDOC 2002, the 6th IEEE International Enterprise Distributed Object Computing Conference, pp. 212-223.

[30]J. Araujo, A. Moreira, I. Brito, and A. Rashid, "Aspect-Oriented Requirements with UML," in Aspect Modeling with UML workshop at the Fifth International Conference on the Unified Modeling Language and its Applications, 2002.

[31]D.M.Simmonds, S.Ghosh , R.B.France, "An MDA Framework for Middleware Transparent Software Development", Proceedings of Real-Time and Embedded Technology and Applications Symposium Workshop on Model-Driven Embedded Systems, Washington, D.C., 2003.

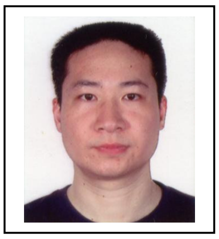
[32]G. Madl, S. Abdelwahed. "Model-based Analysis of Distributed Realtime Embedded System Composition", Proceedings of the 5th ACM international conference on Embedded software, New Jersey, USA , 2005.

[33]R.M.Parizi, A.A.Ghani, "A Survey on Aspect-Oriented Testing Approaches", International Conference on Computational Science and its Applications, Kuala, Lampur, August, 2007,IEEE Computer Society, pp.78-85.

[34]Weis, T., Becker, C., Geihs, K., Plouzeau, N., "A UML Meta-model for Contract Aware Components", Springer LNCS 2185, 2001.

[35] Ritter, T. Born, M. Unterschutz, T. Weis, T. "A QoS Metamodel and its Realization in a CORBA Component Infrastructure",Proceedings of the 36th Annual Hawaii International Conference on System Sciences,2003,pp.318-328.

Authors



Mr. LIU JingYong is a teacher of Faculty of computer, Guangdong University of Technology. He has published over 10 technical papers, some of which are index by EI. His research interests are: real-time and embedded systems, distributed systems, software process and other related topics.