Integration of Non-Functional Properties in Containers

Denis Conan, Erik Putrycz
Institut National des Télécommunications
9, rue Charles Fourier
91011 Évry cedex, France
{Denis.Conan|Erik.Putrycz}@int-evry.fr

Nicolas Farcet, Miguel DeMiguel
THALES
Corporate Research Laboratories
Domaine de Corbeville
91404 Orsay cedex, France
{Nicolas.Farcet|Miguel.DeMiguel}@thalesgroup.com

Abstract

Designers and developers try to build stable and multi-domain software components while the software infrastructure they use keeps evolving and thus being heterogeneous. This can be called "*the business and technical life cycles mismatch*". The paper shows where and why middleware technologies fail to solve the business and technical life cycles mismatch. Then it demonstrates that lots of the nice features of software components are allowed thanks to the concept of container. The container is an entity visible as much during software component development as during execution. Some of its properties must be taken into account in software architectures and design models.

Keywords: Component software, software architecture, non-functional properties, container.

1 Introduction

Nowadays, the following three statements are experimented by all designers, developers, and assemblers of distributed applications :

1. In general, non-functional and technical parts of an application (*e.g.*, transactions, dependability) are much more difficult to design and to implement than business ones. In addition, most of the time, business parts are worth more than non-functional parts[1]. This is why developers should focus on functional parts of their applications.

2. Non-functional and technical parts change more frequently than functional ones. Developers try to build stable and multi-domain software components while the software infrastructure they use keeps evolving and so becoming heterogeneous. This can be called "*the business and technical life cycles mismatch*". A direct consequence is that non-functional parts will be provided by several providers.

3. Non-functional and technical implementations depend on an "*implicit software infrastructure*" (*e.g.*, middleware, middleware services, operating system). In some cases, patterns such as *n*-tiers are promoted to first-class elements of software architectures. In other cases, patterns previously considered as first-class elements of software architectures (*e.g.*, transaction monitor patterns) disappear inside software component frameworks. These patterns get deprecated because not expressed at the appropriate level of abstraction and become design patterns modelling the inside of software components, containers, or middleware services.

The previous statements lead to the following rule of thumb: Separate functional and non-functional parts to attain better time to market and better future evolution. These are important objectives of middleware and software component frameworks such as Microsoft COM+ [Mic01a] and .NET [ECM00], Sun Enterprise

[1] Except of course when product differentiation is done on the basis of the support of non-functional properties.

JavaBeans [DYK99], and OMG CORBA Component Model [OMG99]. Middleware permits to divide an application in distributed parts. Middleware can interoperate between each other and each one can provide non-functional parts[2]. Software component frameworks introduce concepts to ease the assembly of business components together and the usage of middleware services. The most important improvements are the introduction of required interfaces, attribute-based programming, combination of middleware services, packaging, and assembly. More precisely, attribute-based programming provides a powerful and easy way to bind middleware services to software components without affecting their internals. It allows for example semi-transparent configuration of middleware services in a declarative way (possibly by an administrator). Attribute-based programming is done in EJB and CCM technology through XML configuration files associated to software components. In .NET, attributes can be associated to the software components directly in the source code files and are then integrated within the byte-code as meta-data.

The paper shows where and why middleware technologies fail to solve the business and technical life cycles mismatch. Then it demonstrates that lots of the nice features of software components are allowed thanks to the concept of container. The section 2 shows how the software component and the container help in separating the business life cycle and the technical life cycle. The sections 3 and 4 develop the definition of a container and its role in the software architecture and during component software —*i.e.*, design, implementation). Finally, the sections 5 and 6 present related works and list open issues.

2 Motivations and objectives

Even if the middleware technologies help in separating business and non-functional parts, designers and developers face at least two difficulties. 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 in this paper is that during design, the container is also the right place for modelling the combination of non-functional services.

Ideally, the development process including requirements engineering, software architecture, and component software is as depicted in figure 1. The separation of the functional and non-functional concerns begins during requirements engineering. It continues during design with the concept of views. In figure 1, only two views are shown but the software architecture may contain several others. During component software, software components of the logical view of the software architecture are translated into business components. Model elements of the technical view are used to generate or configure the containers of the business components and may translate in new middleware services or technical components.

3 Definition of the term ”container”

We define the term ”container”[3] as *the entity responsible for handling non-functional properties on behalf of a software component.*

Container is an already-used concept in frameworks like EJB and CCM. The container is responsible for handling the two-phase-commit protocol [GR93] and persistency. Transactions managers are very complex to access and the corresponding code can easily be up to 80% of the whole code [Ses00]. Therefore, with a container, the developer only writes the core business logic and the software component framework provides a way to declare the transactional behaviour. Ideally, at the execution level, the container handles

[2]The non-functional parts, when they are provided by a middleware, are called middleware services in the document.

[3]This doesn't refer to the pattern "*Container*" [Dou98]

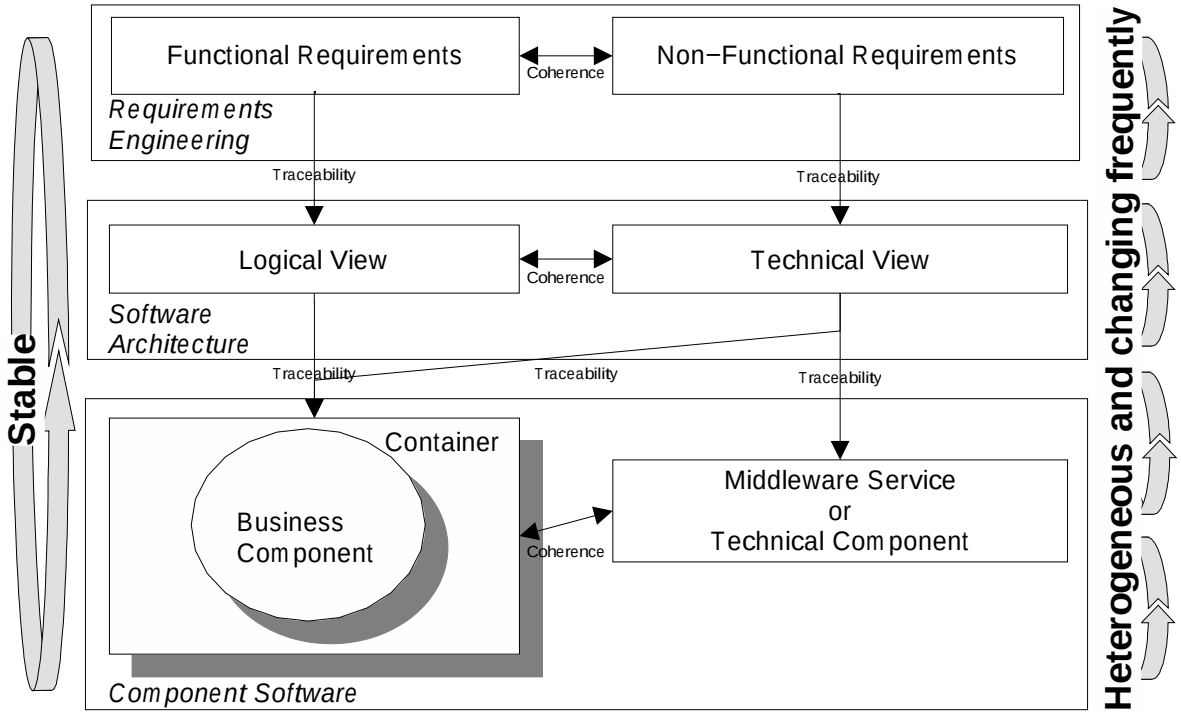

Figure 1: The business and technical life cycles

all the non-functional properties. A huge limitation of the current software component frameworks is that the list of accessible non-functional properties is forged into the framework itself. This is why, in our opinion, the main role of a container is to provide a way to extend this list of non-functional properties.

One of the primary goal of the container is the transparency of the non-functional properties to the software component. The non-functional properties 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. For example, mobile-aware applications that replicate data weakly can provide data-specific procedures to reconcile data after a disconnection followed by a reconnection. In this latter case, the collaboration between a software component and its container must be well-specified in the form of a bilateral contract — *i.e.*, internal[4] offered and required interfaces.

The container is an entity visible during software component development and during execution. In other words, we propose to integrate the concept of container “up to” the software architecture — *i.e.*, in the development meta-model [CCF00]. 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.

To conclude the definition section, the container realizes the combination of non-functional services, except the brokering service which is often provided by the middleware. Used in the software architecture literature [BCK98], connectors realize also non-functional properties, 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. This is a different issue than encapsulating middleware services and is not addressed in this paper.

[4]The term “internal interface” is from [OMG99, DYK99].

4 Component software with containers

The container plays an important role during software architecture just as well as during component software. The figure 2 depicts the big picture showing the role of the container with regard to the application and non-functional services. Three groups of stakeholders are distinguished: the service provider, the container provider, and the application developer. In the figure, there are two services and so two service providers. They are groups of stakeholders since each of them is responsible for the entire development (design, implementation, and delivery) of their software parts. The results of the developments are packages ready to be deployed.

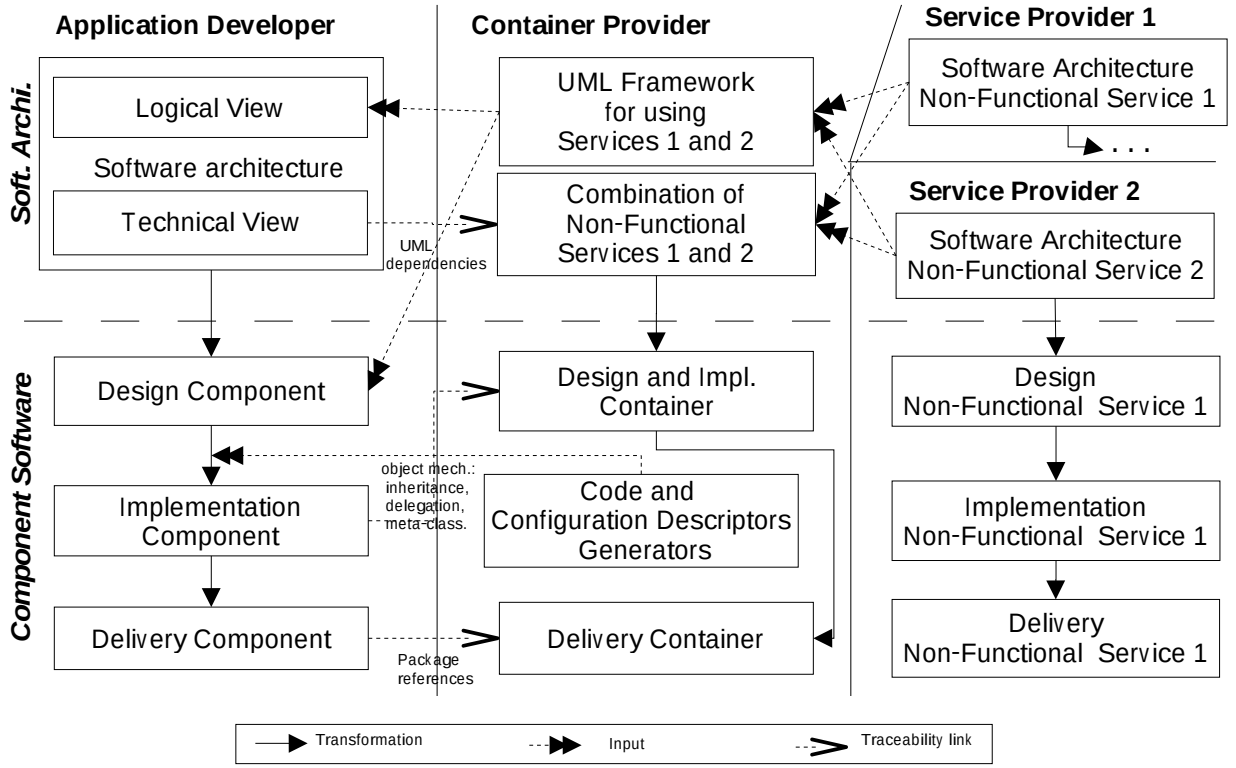

Figure 2: The role of a container in software architecture and component software

The service provider is of course responsible for developing the service. It is not necessary that all the artifacts produced during the development of the service be public. The service provider publishes the software architecture of the non-functional service when it is not a simple one (*e.g.*, when distributed). This allows the container provider to fully understand how it works. The service provider is of course responsible for developing the service. It is not necessary that all the artifacts produced during the development of the service be public.

The container provider prepares artifacts to simplify the modelling 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 (*Cf*. section 4.1). The second task is to help the application developer in transparently integrating the non-functional properties 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 (*Cf*. section 4.2).

The application developer builds the software architecture of the application. The logical view uses the UML framework to stereotype, tag, and constrain the software components. Concerning the logical view, either it shows the combination of non-functional services with architectural patterns or it refers to the models developed by container providers. Clearly, building these two views requires two different skills: business and technical. This justifies the intermediary of the container provider so that the application developer concentrates mainly on business concerns. According to [CCF00], the granularity of software components identified in the software architecture may not correspond to the granularity implied by the chosen software component technology. In this case, the design activity composes software components into "larger" ones or splits software components into "smaller" ones. The resulting software components are for example .NET or EJB software components. This task is an open issue but experiments demonstrate its need. The UML models that show the compositions and the splittings also use the UML framework given by the container provider. Component software continues with code and configuration descriptors generation and the implementation of software components. The code generated may traces back to the code of the container through inheritance for example. Finally, the software components are packaged and delivered to the customer. These packages may not contain all the artifacts included in container packages but only references in the form of URI to these packages.

The sections 4.1 and 4.2 give details of two important issues of the big picture just depicted, namely the use of UML profiles for modelling non-functional properties and the integration of these properties to software components.

4.1 UML framework

In this section, we explain how a UML framework can provide a profile to architects and designers for modelling non-functional dependencies. One or more non-functional properties 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. Non-functional properties 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 properties to a model element. A constraint consists in specifying more semantics as an expression in a designated constraint language (*e.g.*, OCL, programming language, mathematical notations, natural language). Constraints are gaining more and more importance in UML (with the preparation of UML 2.0). 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.

UML stereotype is another extension mechanism simple to define and use. Stereotypes were introduced in UML in order to let the modeller add new meta-elements in CASE tools that do not make their meta-model public and allow the user to modify it. For our problem, the new meta-elements can be closely linked to non-functional properties. For instance, to make a class persistent, simply add a stereotype `<<persistent>>` on it. In addition, it is possible to automatically associate specific tagged values with a stereotype. The main limitation of the stereotype approach is that it's not possible to assign several stereotypes to one model element in UML 1.3 [OMG00]. This can be circumvented by using simple and multiple inheritance of stereotypes. However, UML 1.4 offers the possibility to assign several stereotypes to the same model element. Also note that stereotypes are not at all reserved for modelling non-functional properties; they are for example already used for modelling actors in use cases: *e.g.* clients. Consequently, there exists a risk of confusion that increases with the number of stereotypes.

4.2 Binding non-functional services and software components with a container

In the previous section, we explained how to add non-functional properties to software components in UML architectural and design models. The next step is to use this information during component software (cf. figure 2) to integrate these services into software components with respect to an existing component model.

The integration of new services into a software component framework requires the binding of the business and technical codes. This binding consists of integration points. Integration points are a means to notify an external entity (*e.g.*, a middleware service) of a change in the fields of the software component, a method call issued by / on the software component, a creation / destruction of a software component, or a reference passing between software components. Integration points are inserted either by source-code transformation or through indirection frameworks provided by the software component model. The first way to integrate business code and containers is code transformation used in code generators. If the programming language is object-oriented, the source-code generator can make the most of object mechanisms like inheritance, delegation, and meta-object protocol. Another form of source-code transformation is code weaving like in Aspect/J [LK98]. The aspect code (non-functional properties) is inserted by a code weaver given some configuration parameters. Secondly, this binding can be delayed till the delivery or later by using indirection frameworks. Indirection frameworks like interceptors [SSRB00] provide an external API to insert integration points in a software component. The principle is to specify in configuration descriptors what to intercept and which entity to notify.

The binding of containers with non-functional services consists of configuration, combination, and accesses to non-functional services. Non-functional service access is only done during the execution but the configuration and combination may require tasks at every step of the life cycle (*Cf*. figure 3). The configuration of a non-functional property in the software architecture is sometimes not sufficient for fully configuring the non-functional service. The architect should not be aware of all the low-level configuration details but should leave them to designers and other “following” stakeholders. In addition, to ease the configuration of a non-functional service, its setup can be handled via attribute-based programming. The container controls the combination of the accesses to each non-functional service when they are not orthogonal. Such a combination can be very complex and may not be transparent for some non-functional services but need to be specified in the software architecture. For instance, when providing quality of service and mobility, the contract implied by the quality of service may be temporarily broken when disconnections occur. A current solution is to involve the user in order to know if he/she would accept a lower quality of service and which degradations he/she is willing to accept: *e.g.*, quality of images, fluidity of animated graphics. This requirement impacts the software architecture by introducing interactions with the user.

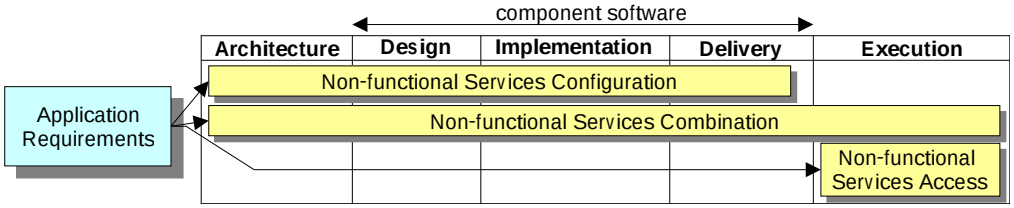

Figure 3: Binding non-functional services in a container

5 Related work

The idea of using containers in software component frameworks to support the management of some non-functional services (security, transaction, persistence, and events) is a basic concept in EJB [DYK99], CCM [OMG99], COM+ [Mic01a] models. But, as we said, the services supported are limited and the container models are based on these services. These standards do not pay enough attention to the introduction of new services and the definition of new container models based on these services.

Separating functional and non-functional parts of an application is also the subject of Aspect Oriented Programming [MLTK97]. AOP aims to decrease the dependencies between the functional components, enhance the reusability and make the source code easier to develop, understand and evolve. The separation of concerns in the source code *la* AOP is a nice way to improve the implementation by separating non-functional and business code. A container may benefit from AOP for its implementation and for the integration to the software component.

Another well-studied subject is the integration of container models in architecture description language, especially UML. [Gre01, MS01] define methods for modelling EJB software components in UML architectures and designs, and introduce UML extensions for that purpose. These extensions are used in the automatic generation of skeletons, configuration files, and assembly. In addition, [Gre01] proposes the introduction of mechanisms for using UML models to describe the content of a software component and configure it. This allows to use UML models for automation and reflection of EJB software components. [Wil99] shows that EJB software components impact the UML structural architecture from two different points of view: i) subsystems and types of architectural components, and ii) layer of services required and provided. [OMG99] specifies the MOF (OMG Meta Object Facility) meta-model of CCM, but does not integrate this meta-model in UML MOF meta-model.

OMG has recently presented a vision called MDA (Model-Driven Architecture) [Sol00] which addresses the problem of building logical architectures with few dependencies on infrastructures, thus trying to address the business and technology life cycles mismatch. The MDA approach aims at being able to create platform-independent application models (PIM) that can be later converted into platform-specific models (PSM). MDA relies for that mostly on UML and its extension mechanisms. Typically, UML profiles are used to extend and to constrain UML to describe PIM and PSM. EDOC is the major UML profile to describe component-based PIM with the help of UML profiles for PSM (*e.g.*, EJB and COM+ UML profiles).

Recently some work has been developed for the integration of new technical services in software component frameworks. The most advanced developments in this field are .NET extensible object contexts. As explained in [Mic01b], "a context is an ordered sequence of properties that define an environment for the objects resident inside it. Contexts get created during the activation process for objects that are configured to require certain automatic services such as synchronization, transactions, just-in-time activation, security etc. (...) A new object's context is generally chosen based on meta-data attributes on the class." The usage of custom attributes applied to classes is then the step to go to allow the deployment of custom non-functional services and associated policies. [KC99, WKS00] propose solutions for the automatic reconfiguration of software components to provide fault-tolerant and real-time support: [WKS00] introduces the QoS Container Adaptor concept. This type of container is based on scheduling and QoS services to provide QoS support to business components. The objective is to provide support for the integration of QoS properties of real-time and messaging CORBA services, allowing a flexible, transparent, and adaptive configuration. The container manages certain QoS properties of component implementations such as memory, bandwidth, concurrency, dependability, and power consumption. It supports end-system dynamic configuration of QoS and developers can defer the selection of QoS requirements until run-time. They do not analyze the impact of new container model in software architectures.

6 Conclusion

We have shown that the integration of non-functional properties into containers significantly raises the abstraction level of middleware, and consequently of software architectures built onto them. Configuration and usage of middleware services that had to be imperative, global, and interleaved in application code with middleware are done locally through attributes declaration — *i.e.*, at a meta level — in software component frameworks. Attribute-based programming can be seen as a major advance to application development. It may well be further emphasized as technology providers and standard organizations will improve the extensibility mechanisms of current software component frameworks.

Open issues can be classified into infrastructure and modelling issues. Concerning the infrastructure, there are still complex services that are to be seamlessly integrated into software component frameworks

such as QoS handling, fault tolerance, and real-time. Concerning the modelling, new architectural and design patterns have to be proposed. The support for the business and technical separation should be integrated into development processes. The impact of late binding should also be further studied.

References

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 1998.

[CCF00] D. Conan, M. Coriat, and N. Farcet. A Software Component Development Meta-Model for Product lines. In *Proc. ECOOP '00 Workshop on Component Oriented Programming*, Nice, France, June 2000.

[Dou98] B. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, second edition, 1998.

[DYK99] L. DeMichiel, L. Yalinalp, and S. Krishnan. *Java 2 Platform Enterprise Edition Specifications, v2.0*. Sun Microsystems, 1999.

[ECM00] ECMA. Common Language Infrastructure (CLI). ECMA/TC39/TG3/2000/2 part 1, 2, 3, and 4, http://www.ecma.ch, October 2000.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gre01] J. Greenfield. Uml/ejb mapping specification. JSR #000026, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_026_uml.html, March 2001.

[KC99] F. Kon and R. Champbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proc. 5th Conference on Object-Oriented Technologies and Systems, USENIX' 1999*, 1999.

[LK98] C. Lopes and G. Kiczales. Recent Developments in AspectJ. In *Proc. of ECOOP'98 AOP Workshop*, 1998.

[Mic01a] Microsoft. Microsoft developer network. http://msdn.microsoft.com/, June 2001.

[Mic01b] Microsoft. .NET Framework SDK Beta 1 documentation. http://msdn.microsoft.com/net, June 2001.

[MLTK97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming workshop report. 1997.

[MS01] V. Matena and B. Stearns. *Applying Enterprise JavaBeans*. Addison-Wesley, 2001.

[OMG99] Object Management Group. *Corba Component Model Specifications*, July 1999.

[OMG00] OMG. *Unified Modeling Language Specifications 1.3*. OMG, March 2000.

[Ses00] R. Sessions. *COM+ and the Battle for the Middle Tier*. Wiley, 2000.

[Sol00] R. Soley. *Model Driven Architecture*. Object Management Group, November 2000.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Bushmann. *Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. Wiley, 2000.

[Wil99] C. Wilson. *Application Architectures with Enterprise JavaBeans Component Strategies*. Addison-Wesley, August 1999.

[WKS00] N. Wang, M. Kircher, and D. Schmidt. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *Proc. COMPSAC' 2000*, 2000.