Arctic Beans Containers Composition of Non-functional Services Using Composition Filters

Jie Yang
University of Tromsø
Department of Computer Science
Tromsø Norway
jie@cs.uit.no

Gordon S. Blair
Lancaster University
Computing Department
Lancaster, UK
gordon@comp.lancs.ac.uk

Anders Andersen
University of Tromsø
Department of Computer Science
Tromsø, Norway
aa@cs.uit.no

ABSTRACT

It is becoming clear that modern middleware platforms must provideboth deploy-time configuration and run-time reconfiguration to accommodate rapid changing requirements and also to be able to operate in dynamic environments. J2EE is a key example of a middleware architecture that supports enterprise applications via its Enterprise JavaBeans (EJB) component model. EJB provides limited configurability in terms of a fixed set of non-functional middleware services at deployment-time (via a declarative deployment descriptor). However, EJB along with other related enterprise architectures generally do not provide enough support for re-configuration or evolution. At best, there is limited support in some platforms for replacing or updating particular services. This paper discusses the design of configurable and re-configurable middleware architecture and also the key role of separation of concerns for such platforms. The paper also describes the Arctic Beans component model which uses the Composition Filters model to capture such concerns and also support their safe composition. The paper also explains how this model can be used to construct an Arctic Beans container, in the style of EJB. The main contribution of the paper is to demonstrate that adaptable middleware platform can be developed using separation of concern technologies, specifically the composition filters model.

1. INTRODUCTION

As pointed out in [1], most existing middleware platforms are developed with an underlying black-box philosophy that hides the underlying middleware architecture and offers fixed services to their users. Based on this, it is argued that "next generation middleware platform should be *configurable*, to meet the needs of a given application domain, *dynamic reconfigurable*, to enable the platforms to respond to changes in their environment, and *evolvable*, to meet the needs of changing platform design" [1].

This argument applies to the full range of middleware platforms available. In this paper, we are particularly concerned with the arguments for flexibility in the context of the Java 2 Enterprise Edition (J2EE), which is designed to support the rapid development of enterprise applications with particular focus on client/server and multi-tier middleware architectures. More specifically, we are mainly concerned with the Enterprise Java Beans (EJB) technology. EJB provides limited configurability in terms of a fixed set of middleware non-functional services at deployment-time (via a declarative deployment descriptor). However, EJB along with many other related enterprise architectures generally do not provide enough support for re-configuration or evolution. At best, there is limited support in some platforms for replacing or updating particular services.

Complexity is a key issue in contemporary middleware platforms and it is this complexity which often hampers attempts to provide more flexibility. In particular, in such platforms, it is necessary to consider and eventually integrate a number of potentially overlapping concerns (e.g. security, transactional behavior, etc.). Given this, separation all concerns is crucially importance in software systems in general, but particularly in the area of middleware. Good support for separation of concerns can reduce complexity, improve reusability and simplify evolution. However, programming language mechanisms and associated platforms typically do not support separation of concerns very well, resulting in the problems of *tangling* and *scattering* that significantly complicate software maintenance, evolution, integration, and reuse [2]. Adaptive programming [3], AspectJ [4], Hyperspace [5] and the Composition Filters (CF) model [6] are examples of techniques that have recently been proposed to address this problem. In this paper, we are particularly interested in the Composition Filters (CF) model as a modular extension of the traditional object model that provides both intra-class and inter-class composition of crosscutting concerns [6, 7]. Crosscutting concerns are expressed modularly and orthogonally, thus the adaptability and reusability of concern are increased. This analysis applies in particular (but not exclusively) to non-functional aspects of middleware. Each individual kind of non-functional service may be considered a separate concern dimension [5] and modeled separately [8, 9]. AspectJ2EE [10] and JBoss [11] are two examples that achieving better flexibility and extensibility of middleware platform with techniques of separation of concerns.

In this paper, we focus on the design of configurable and re-configurable middleware with particular attention to EJB-like platforms (although many of the arguments generalize to other enterprise architectures and indeed more general styles of middleware). In particular, we investigate the role of composition

filters in providing the necessary separation of concerns and structure to support the dynamic properties we seek. A key motivation for investigating this approach (over for example other aspect-oriented technologies) is the ability to reason about composition and the subsequent correctness of middleware instances. As an added dimension, we also consider the integration of an underlying reflective component technology providing a more principled approach to introspection and adaptation of the resulting structures.

More specifically, we present the Arctic Beans component model, which employs the Composition Filters model for expressing the concerns relating to non-functional services (and also potentially in the future other concerns, e.g. relating to self-* properties). We also present the Arctic Beans container architecture as a particular instantiation of the more general architecture offering configurability and re-configurability of those services in the container.

The rest of paper is structured as follows. Section 2 provides the necessary background information on composition filters and component technology. Section 3 then describes the design of the Arctic Beans component model, and the Arctic Beans container architecture. Finally, future work and conclusions are discussed in section 4.

2. BACKGROUND

2.1 Composition Filters Model

The Composition Filters (CF) model [6, 7] enhances the conventional object model by intercepting and manipulating all incoming and outgoing messages. It contains three selective parts: filter modules, superimposition mechanism and the implementation object.

The *filter modules* specify how messages should be handled in terms of the concept of filters and the semantics of filter types. More specifically, the filter module part contains input filters and output filters that manipulate respectively incoming and outgoing messages of an object. The semantics associated with acceptance and rejection of a message depends on the semantics of the filter type (Dispatch, Error, Wait, Meta) of the filter. Further details of the filter types can be found in [6, 7].

The *superimposition mechanism* specifies selected behaviors and the location where the behavior should be superimposed. It contains a selectors part and other sections. The selectors part specifies the location of the superimposition through a number of join point selectors. A number of sections specify selection of object behaviors that are superimposed upon the locations declared by the selectors.

The *implementation object* is an enhanced conventional object that offers an interface of two types of methods: regular methods for functional behavior of the object, and conditional methods (also called conditions) for filters to test the state of the object.

The CF model can be used to model separation of concerns with filter modules for the concerns specification, the implementation part for the implementation of the underlying behavior, and the superimposition mechanism for composition of crosscutting concerns.

A CF class can represent a concern that crosscuts its objects by specifying composition of the concern with its filter module set; this is also called intra-class modeling of crosscutting concerns. Crosscutting concerns over several CF classes are modeled by encapsulating each concern within a CF class and specifying composition of the software artifacts through the superimposition mechanism.

2.2 JBoss

JBoss is an extensible server [11, 12], which allows the user to extend middleware services by dynamically deploying new components into the JBoss server at run time. There are two kinds of components in the JBoss system: EJB components and JBoss service components:

- EJB components are Java objects that conform to the Enterprise JavaBeans (EJB) architecture [13] and thus implement a set of Java interfaces for either remote or local client. EJB components are used to implement the business logic for the appropriate application domain. JBoss containers then provide an execution environment for such components.
- JBoss service components are Java objects that conform to the JBoss component model convention and expose a management interface to address issues like services lifecycle and dependencies between services to their client. The JBoss service component model extends and refines the JMX service component model (MBeans) [14]. JBoss service components are used to implement middleware services for a JBoss-based system. There are two kinds of JBoss service components: Service MBeans and Deployable MBeans. Service MBeans add service life-cycle operations to the original management interfaces. Deployable MBeans encapsulate service MBeans in a deployment unit according to an EJB-like convention, and also manage dependencies between those service MBeans.

2.3 The JBoss EJB Container

In a JBoss [11] system, a generalized EJB container manages the deployed EJB component and provides the EJB component with pluggable middleware non-functional services such as instance pooling, instance caching, persistence, security, and transactions. The abstraction of an EJB container is realized by a container MBean together with the set of plug-ins. There are two kinds of container plug-ins: (a) well defined plug-ins are used to implement specific services, like bean instance pooling, bean instance caching, and management of bean persistence, and (b) server-side interceptors are used to implement non-functional services of the container.

EJB containers are JBoss service components themselves. The JBoss EJB container provides both base-level interfaces and meta-level interfaces. Base-level interfaces are the interfaces of the EJB component. In contrast, the meta-level interfaces are the MBean management interfaces of JBoss service components.

At deployment time, the EJB container is configured by specifying all the information required to create the container in XML files. JBoss provides a default container configuration for the standard EJB components with a global configuration file (`standardjboss.xml`), and an alternative configuration with a local configuration file (`jboss.xml`) optionally included by a given EJB component. Configuration of non-

functional services is supported by manually changing the server-side interceptors in the configuration file of the EJB container.

3. OVERALL DESIGN

3.1 The design of Artic Beans Component

Arctic Beans components are software components used to encapsulate middleware non-functional services. They are built on top of two models: the CF model and the OpenCOM component model [15]. By combining those two technologies, the Arctic Beans component model provides better support for dynamic adaptation and evolution of both Arctic Beans components and middleware platforms constructed from this technology. In essence, OpenCOM provides the necessary level of openness and the CF approach associated support for (correct) composition.

As mentioned in the previous section, middleware non-functional services may be designed as separate concerns to maintain conceptual decomposition and provide better reusability. In our design, a concern is represented by the CF model, because it can express crosscutting concerns with modularity and orthogonality [6].

The Arctic Beans component model applies the CF model as follows:

- The CF model supports the assembly of components of various (potentially heterogeneous) component models. Components of different component model may also coexist in the Arctic Beans component, and provide different functionalities for the containing Arctic Beans component.
- Composition of non-functional concerns is facilitated by the superimposition mechanism. The CF model provides composition of crosscutting concerns at the inter-class level; various concerns are composed through superimposition mechanism. As non-functional services are represented as concerns, configuration and reconfiguration of non-functional services can thus be facilitated by inter-class level concern superimposition.
- Construction and reconstruction of individual concern is supported by unification of functionalities of sub components. The CF model provides also composition of crosscutting concerns at intra-class level; behaviors of sub-components are represented as filter modules and composed together within a CF object and the associated superimposition mechanism. This feature of the CF model is capable of composing the functionality of internal components together, and recomposing them later if needed. This has the nice feature of not just separating underlying concerns but also of having a clean separation of concerns between such features and their composition (cf. research on co-ordination and components).

In order to support dynamic adaptation of Arctic Beans components, Arctic Beans components need to provide meta-interfaces for architecture introspection and adaptation. For this, we adopt the OpenCOM reflective component technology. OpenCOM is a minimal, lightweight component technology designed for the construction of low level systems software including middleware and embedded systems. A key feature of OpenCOM is its intrinsic support for reflection, through a number of meta-interfaces. These interfaces support both introspection and adaptation at the level of individual components and their interfaces and also crucially at the level of software architectures [15]. An Arctic Beans component contains an OpenCOM component as an internal component and delegates invocation of corresponding reflective operation to the appropriate meta-interfaces of this underlying OpenCOM component.

As shown in figure 1, the structure of an Arctic Beans component consists of three filter modules (the ImportService filter module, the InjectMetaInterface filter module and the ExportService filter module), a superimposition mechanism, and an implementation object.

The main functionality of the Arctic Beans component is to impose middleware non-functional services. Such services are firstly implemented as JBoss service components, and then encapsulated in the Arctic Beans component. The ImportService filter module declares a JBoss service component with the implemented service to be included in the Artic Beans component. With the help of the superimposition mechanism, the JBoss service component is incorporated in the Arctic Beans component instance.

As mentioned above, an Arctic Beans component provides reflective functionalities including, in particular, architectural introspection. This functionality is included into the Arctic Beans component by the InjectMetaInterface filter module. Together with the superimposition mechanism, the InjectMetaInterface filter module specifies reflective functionalities of an OpenCOM component to be included in the Arctic Beans component instance.

An Arctic Beans component need also specify a destination (base) program where the services provided by this Arctic Beans component are woven into, in other words, which objects that are allowed to invoke/access those services and which kind of services are exported to those objects. The ExportService filter module specifies the kind of services to be exported. This specification contains all the interfaces required, including meta-level interfaces of OpenCOM and management interfaces of the JBoss service component, as all those interfaces are required for the destination program to function properly. In other words, the ExportService filter module specified all the interfaces and sub components needed to form a component assembly. The specified interfaces conform to the OpenCOM component model specification.

The superimposition part of the CF model groups different superimposition tasks together, where different task can have different selected destination programs and behaviors. An object is aware of the imported behavior superimposed on it at the instantiation time of the object. For the case of Arctic Beans components, there is a specific superimposition task that cooperates with each filter module. The selected behavior is added either to the Arctic Beans component instance itself (e.g. through the ImportService filter module) or the selected EJB container type. At instantiation time, the EJB container instance is aware of the available non-functional services and configures them accordingly.

An Arctic Beans component also contains internal (sub) components as its building blocks. Those sub components can be components of different component models, for example, JBoss service components or OpenCOM components and they may coexist (usefully) within the containing Arctic Beans component.

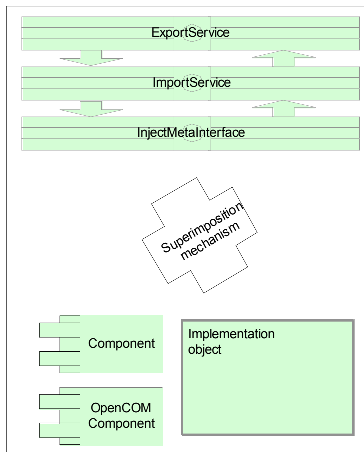

Figure 1: Arctic beans component.

A sub component can be included in an Arctic Beans component in two different ways: (a) when a sub component is not shared by other CF components, it is located within the containing Arctic Beans component; (b) when a sub component is shared by other CF components, it is located outside of the Arctic Beans component and is available to the Arctic Beans component via its object reference.

In order to achieve better harmony and to provide cohension, an Arctic Beans component needs to handle the issues of sub components interaction. Interactions between sub components are handled at the implementation object.

An Arctic Beans component provides two kinds of interfaces: base-level interfaces and meta-level interfaces:

- The base-level interfaces are the union of available interfaces of its sub components and implementation object thus providing the main functionality of the Arctic Beans component. For example, in the case of an Arctic Beans component that realizes a transaction model, the JBoss service component interfaces that realizes transaction services all belong to this group.
- The meta-level interfaces provide reflective capabilities for the component. They are a union of available meta-level interfaces of its sub-components and implementation object.

For both base-level and meta-level interfaces of Arctic Beans components, operation invocations are handled in the same way. An invocation message first arrives at the filters, it then passes along the filters until a particular filter accepts it and dispatches it for execution.

As an illustration of the approach, we provide an example of a sample specification of a transaction service as shown in figure 2.

The `ImportService` filter module specifies the transaction service `TA` with its methods `StartTransaction()`, `StopTransaction()` and `AbortTransaction()`. The invocation of the service is then delegated to the transaction service through a Dispatch filter `delegate`. The `ImportService` filter module is further composed into the concern instance by the superimposition mechanism.

The `InjectMetaInterface` filter module attaches meta-level interfaces to the concern through an OpenCOM component `SubComponent`. The functionality included into the concern are `IMetaArchitecture()`, `IMetaInterface()`, `ILIfeCycle()` and `IConnection()`. The `InjectMetaInterface` filter module is also composed to the concern instance by the superimposition mechanism.

The `ExportService` filter module specifies all the interfaces included to export this transaction service. The interfaces included are all operations for the transaction service and all operations that realize the reflective functionality. The concern `ABCompTA` is defined as a shared object and the `ExportService` filter module holds an object reference to it. The superimposition mechanism superimposes the concern to all containers of type `ABContainer`.

3.2 The Arctic Beans Container

An Arctic Beans container is an example of an Arctic Beans component that provides an execution environment for EJB style components. It also supports adaptation of non-functional services of the EJB container.

The design of Arctic Beans container needs to handle the following issues: (1) Obtaining available filter modules that implement non-functional services. (2) Composing those filter modules together in the specified order. (3) Maintaining the integrity of the container structure.

As every non-functional service is specified as an ExportService filter module of an Arctic Beans component and superimposed into the Arctic Beans container, the available filter modules are made available to the container in an arbitrary order [16].

Filter modules superimposed into the Arctic Beans container implement all the interfaces of an OpenCOM component. It is actually an OpenCOM component in the form of a CF filter module, and can thus be connected together as with any OpenCOM component. An associated OpenCOM component framework [15] maintains an internal structure of interconnected OpenCOM components and provides functionality for introspection and change of the internal structure. An Arctic Beans container uses this component framework structure to connect those non-functional services.

Integrity of the container structure is maintained by a centralized composition rules component that implements an IAccept interface. The component framework sends invocation to the IAccept interface when changing its internal structure and the associated object must approve of the changes.

An Arctic Beans container provides configuration and reconfiguration of non-functional services as follows:

- Connection of filter modules using the OpenCOM component framework.
- Configuration of non-functional services by container composition of internal objects at instantiation-time. Since the Arctic Beans container is based on the CF model, elements of the CF model are combined easily with the feature of intra-class composition, that is, with the CF message manipulation language programming. Therefore, configuration of non-functional services of Arctic Beans container is implemented directly using CF message manipulation programming.
- Reconfiguration of non-functional services with updated information at run-time. An Arctic Beans container holds object references to two centralized components with information of all the available filter modules of the container, and the general composition rules. When there is a need for dynamic reconfiguration, the container can be reconstructed by repeating the container composition process again with the updated information.

As shown in figure 3, an Arctic Beans Container contains a ContainerMetaInterface filter module, a superimposition mechanism and the implementation object.

An Arctic Beans container uses a designated filter module, ContainerMetaInterface, to specify the provided functionality and participating sub-components: the original EJB component, OpenCOM component frameworks and associated Accept components, and the Repository component. The superimposition mechanism weaves the filter modules into the Arctic Beans container. The implementation object connects the IAccept receptacle of the component framework to the IAccept interface of the Accept component, and implements the provided operation IContainerComposition() by combination and reconciliation behaviors of involved sub-components.

Sub-components involved in the ContainerMetaInterface filter module are defined as either internals or externals. The original EJB container component and the OpenCOM component framework are designed as internals, and thus available within the container. The OpenCOM Accept component and the Repository are designed as externals; they are located as outside the Arctic Beans container within the same JBoss server, and are accessible through object references.

The original EJB container component is the original EJB container that configures the non-functional services from the deployment descriptor file. It contains user specified non-functional service order of the EJB component at deployment time of the EJB component, and implements the set of interfaces that conform to the EJB component model.

The repository provides information on all the filter modules superimposed into the Arctic Beans container class. The OpenCOM component framework carries out the actual task of non-functional service connection using OpenCOM component

```
concern ABCompTA begin

filtermodule ExportService begin
    externals
        TAService: ABCompTA;
    methods
        StartTransaction();
        StopTransaction();
        AbortTransaction();

        IMetaArchitecture();
        IMetaInterface();
        ILifeCycle();
        IConnections();
    inputfilters
        startService: Meta =
        {[*]TAService.[StartTransaction|
        StopTransaction|AbortTransaction]};
        dispMetaOperation: Dispatch =
        {IMetaArchitecture,IMetaInterface,
        ILifeCycle,IConnections};
end filtermodule ExportService;

filtermodule ImportService begin
    externals
        MyTA: TA;
    methods
        StartTransaction();
        StopTransaction();
        AbortTransaction();
    inputfilters
        delegate: Dispath = {inner.*,
        MyTA.[StartTransaction|
        StopTransaction|AbortTransaction]}
end filtermodule ImportService;

filtermodule InjectMetaInterface begin
    internals
        SubComponent: OpenCOMComponent;
    methods
        IMetaArchitecture();
        IMetaInterface();
        ILifeCycle();
        IConnections();
    inputfilters
        disp: Dispatch = {inner.*,SubComponent.*}
end filtermodule InjectMetaInterface;

superimposition begin
    selectors
        allABContainers =
        {*->select(oclIsTypeOf(ABContainer) ) };
    filtermodules
        allABContainers <- ExportService;
        self <- ImportService;
        self <- InjectMetaInterface;
end superimposition;

implementation in Java;
class ABCompTAClass
{ ......
}
end implementation;
end concern ABCompTA;
```

Figure 2: Transaction service implemented as concern ABCompTA.

reflective capabilities. The OpenCOM Accept component verifies the composed structure.

```
concern ABContainer begin

filtermodule ContainerMetaInterface begin
    internals
        OrgContainer: EJBContainer;
        MyCF: OpenCOMCF;
    externals
        CompositionVerification:
            OpenCOMAcceptComponent;
        MyFilterModules: Repository;
    methods
        IContainerComposition();
    inputfilters
        disp: Dispatch = {inner.*,OrgContainer.*}
end filtermodule ContainerMetaInterface;

superimposition  begin
    filtermodules
        self <-ContainerMetaInterface;
end superimposiiton;

implementation in Java;
class ABContainerClass
{
    public int
IConatinerCompositon(MyFilterModules,
                      OrgConatiner,
                      ExtraConstraints,
                      MyCF,
                      CompositonVerification)
    {
    ......
    }
}
end implementation;
end concern ABContainer;
```

Figure 4: Artic Beans container

The resulting Arctic Beans container is an assembly of software components that has several filter modules, each related to a particular non-functional service, a ContainerMetaInterface filter module for composition of sub-components of those filter modules, and an implementation object.

An Arctic Beans container provides two kinds of interfaces: base-level interfaces and meta-level interfaces. Base-level interfaces are the set of interfaces of the EJB component. The meta-interface provides the IContainerComposition() operation for architecture introspection of the container.

An example of an Arctic Beans container is shown in figure 4. The filter module `ContainerMetaInterface` specifies an extra sub-component needed for container composition. The original EJB container and OpenCOM component framework are made available as internals `OrgContainer` and `MyCF`; the Accept component of OpenCOM and the Repository as externals `CompositionVerification` and `MyFilterModules`. The meta-level operation `IContainerComposition()` is declared here and implemented at the implementation object.

When a message invokes one of the operations implemented by original EJB container, it is delegated to `OrgContainer` by the dispatch filter `disp`. Invocation of `IContainerComposition()` is delegated to the implementation object.

4. EVALUATION AND CONCLUSION

It has been pointed out that next generation middleware should be both more configurable and reconfigurable and that separation of concerns is a key factor in supporting such features. The selection of an appropriate technology is therefore important for middleware designers. AspectJ2EE is a new aspect language that is used to handle composition of services to user applications in the context of J2EE at deployment time. Composition of services is performed by application assembler in AspectJ2EE. JBoss 4.0 is another example that uses separation of concerns technology for non-functional services composition at deployment time with JBoss AOP framework.

In this paper, we have studied the role of Composition Filters in providing the necessary level of separation of concerns to deal with the complexity of middleware, and have demonstrated how non-functional concerns as well as container architectures can benefit from the expressibility of the Composition Filters approach. Through the design of our Arctic Beans container architecture, we are convinced that adaptable middleware platform can be achieved by using separation of concerns provided by the Composition Filters model and reflection provided by OpenCOM component model.

The Composition Filters model is used to both unify the functionalities of sub-components within individual concerns, and superimpose concerns into the target program (such as EJB containers) at component instance level. By combining OpenCOM component model and Composition filters model, the ability for introspection and adaptation of service components and expressiveness of service behavior are nicely integrated. We do this combination at two levels, the individual concern level and the container architecture level. The flexibility and adaptability can therefore be provided at both levels.A prototype implementation of the architecture has been prepared using JBoss, v 4.0, together with the Compose* tool.

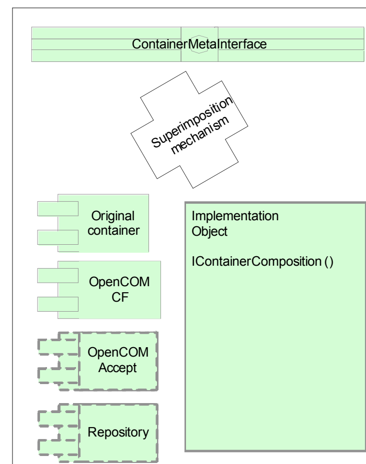

Figure 3: Arctic beans container.

This work is part of the larger Arctic Beans project at Tromsø and ongoing projects are looking at the application of such architectural concepts to dealing with the non-functional concerns of transactions and security. Attention is also being given to the role of context in supporting adaptation in container architectures and on the application of this technology to mobile computing.

5. ACKNOWLEDGMENTS

The research described in this paper is funded by Telenor (R&D) in Tromsø, Norway, as part of Arctic Beans project. The Arctic beans project is also funded by the Norwegian Research Council (IKT2010).

6. REFERENCES

1. Blair, G.S., et al., *The design and implementation of Open ORB 2.* IEEE Distributed Systems Online, 2001. **2**(6).
2. Elrad, T., et al., *Discussing aspects of AOP.* Commun. ACM, 2001. **44**(10): p. 33-38.
3. Lieberherr, K., D. Orleans, and J. Ovlinger, *Aspect-oriented programming with adaptive methods.* J Commun. ACM, 2001. **44**(10): p. 39-41.
4. Kiczales, G., et al., *An Overview of AspectJ.* Lecture Notes in Computer Science, 2001. **2072**: p. 327-355.
5. Rouvellou, I., S.M.S. Jr., and S. Tai. *Multidimensional Separation of Concerns in Middleware*. in *In Proceedings of the Multidimensional Separation of Concerns in Software Engineering,*. 2000: published in conjunction with the 2000 International Conference on Software Engineering (ICSE 2000), 4-11 June, 2000.
6. Bergmans, L. and M. Aksit, *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, 2001. **vol. 44 No. 10**: p. 51-57.
7. Bergmans, L. and M. Aksit, *Principles and Design Rationale of Composition Filters*, in *Aspect-Oriented Software Development*, R.E. Filman, et al., Editors. 2005, Addison-Wesley. p. 63-96.
8. Chung, L., et al., *Non-Functional Requirements in Software Engineering*. Series: The Kluwer International Series in Software Engineering. Vol. Vol. 5. 1999: Springer. 476 p.
9. Lamsweerde, A.v. *Goal-Oriented Requirements Engineering: A Guided Tour*. in *Proc. RE'01 - 5th IEEE International Symposium on Requirements Engineering*. 2001. Toronto: IEEE CS Press.
10. Cohen, T. and J.Y. Gil. *AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework.* in *18th European Conference on Object-Oriented Programming*. 2004. Oslo: Springer-Verlag.
11. Fleury, M. and F. Reverbel. *The JBoss Extensible Server*. in *Middleware 2003 ACM/IFIP/USENIX International Middleware Conference*. 2003. Rio Othon Palace Hotel, Rio de Janeiro, Brazil: Springer-Verlag.
12. Reverbel, F., B. Burke, and M. Fleury. *Dynamic Deployment of IIOP-Enabled Components in the JBoss Server*. in *In Component Deployment: Second International Working Conference (CD 2004)*. 2004. Edinburgh, UK, . Springer-Verlag.
13. Sun Microsystems, *Enterprise JavaBeans Specification, Version 2.1*. 2003.
14. Sun Microsystems, *Java Management Extensions - Instrumentation and Agent Specification, v1.1*. 2002.
15. Grace, P., G.S. Blair, and S. Samuel. *ReMMoC: A Reflective Middleware to support Mobile Client Interoperability*. in *International Symposium on Distributed Objects and Applications (DOA)*. 2003. Catania, Sicily (Italy).
16. Vinkes, C., *Superimposition in the Composition Filters Model*, in *Electrical Engineering, Mathematics and Computer Science*. 2004, University of Twente: Twente. p. 164.