

Component-Based Software Development with Aspect-Oriented Programming

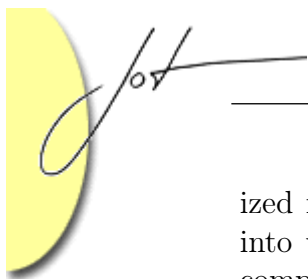
Michael Eichberg, Departement of Computer Science, Darmstadt University of Technology, Germany

Middleware for component-based software development already provides some separation of concerns between the *components* implementing the business functionality and the *component environment* implementing the infrastructural services. However, the implementation of the services is usually not modularized, making it hard to adapt the platform to application specific needs, to exchange services to cope with changing requirements or to use it on different devices. Also, mapping components to objects results in code where the crosscutting concerns encapsulated in the middleware show up at several places, complicating the programming model and making the component code dependent on the used component framework. In this paper an approach to solve these problems based on the ideas of aspect-oriented programming is proposed.

1 INTRODUCTION

Current middleware for component-based software development (CBSD) based on the Enterprise Java Beans (EJB) or CORBA Component Model provide good separation of concerns between the business logic (implemented by the components) and the technical infrastructure needed to run the business logic (implemented by the container). The container implements *middleware services* e.g., to authenticate users, to make an application remotely accessible, to provide transaction handling, etc., and invokes such services at appropriate points during the execution of the business logic in a transparent way. Without the dedicated support by the component middleware the implementation, respectively the invocation and orchestration of middleware services, would be scattered around and tangled with the business logic. Component middleware modularizes this crosscutting.

However, we can observe two main problems with this modularization. The first problem concerns the separation of the business logic from the middleware services. Current approaches force the developer to map component concepts onto language constructs designed to express lower-level concepts such as objects (i.e., Java classes and interfaces in EJBs), often involving coding conventions. The encapsulated middleware services appear at several places in the application code like the tip of an iceberg. This complicates the programming model and defeats the benefits of static type checking. A more direct support for the concept of distributed components in the programming model would make the business logic modeled in the components more maintainable and will foster the reusability. Second, middleware services while well separated from the business logic are themselves generally not well modular-



ized from each other. A modularization of the services provided by the container into well encapsulated and decoupled modules is important to support adaptable component environments that can be tailored to specific application's needs. The vision is a virtual container reified per component type / application out of a set of services that are composed on-demand.

Motivated by these observations, an approach is proposed to apply aspect-oriented programming (AOP) to the design of middleware frameworks, basically modeling each service in a separate aspect. The platform that is proposed, is named Alice. It uses standard Java 1.5 annotations [1] to specify a component's properties. At deployment-time these properties are evaluated and used by the services (aspects) to select join points. One can think of annotations as a lightweight means to extend object-oriented languages with component concepts.

In the next section we outline problems of current AOP approaches. Based on this analysis the concept of Alice is presented. A short summary and discussion of future work ends the paper.

2 DEFICIENCIES OF CURRENT AOP APPROACHES

In this section, we basically discuss the deficiencies of AspectJ [8] with regard to support for modularizing middleware services. AspectJ was chosen because it is the most mature AOP approaches currently available. The author is well aware that a large number of approaches (e.g.: [2, 4, 7, 14, 15]) exist that also try to address these problems, but unfortunately a detailed discussion is out of scope for this paper. However, the problems discussed in the following basically apply to the other approaches as well.

AspectJ provides a good starting point as an approach for addressing the deficiencies of current component middleware outlined in the introduction. In principle, it can be used to modularize individual middleware services, while not constraining the development of components in any particular way, i.e. the component model is not fixed. AspectJ is already successfully used in many projects [3, 10, 17], especially for the implementation of infrastructural services [11, 13, 18]. However, as discussed in the following, AspectJ still lacks some important features: **1.** Support for sophisticated code generation / transformation is required. E.g., to implement a service like passivation [16] it is necessary that all references to the component can be fully controlled by the service, otherwise, the service could be bypassed leading to a faulty runtime behavior. For this purpose, a proxy class can be generated that ensures that the component does not pass a reference to itself (`this`) to any other component - the proxy object is passed instead. **2.** Certain services (aspects) are applicable to a class only if the latter has certain properties. E.g., in order for a passivation service to be applicable to a class, all fields have to be serializable. Hence, it is necessary to check the properties of a class before applying an aspect. Though, AspectJ can be used to enforce design properties [9, 12] the support to enforce structural properties is too limited [5]. **3.** Further, AspectJ provides no standard way to

express the interaction of a class with an aspect. When aspects are used to modularize middleware services, an interface that allows the business logic to interact with an aspectualized middleware services is needed, e.g., imagine a shopping cart or an order processing component that - as part of its business logic - generates an invoice or an order confirmation starting with "Dear Ms/Mr XYZ,...". In order to do so, the component needs to access the data of the user making the order, but the identity of the user performing the transaction is usually known to the authentication service - a typical crosscutting concern implemented as an aspect. Hence, we need to be able to access the authentication service from the component to get the user. 4. Also the expressiveness and abstraction capabilities of the pointcut language is too limited. In [6], Gybels and Brichau discuss the problem of *arranged patterns*, roughly speaking, referring to the problem that often pointcut definitions are based on naming conventions. This leads to a coupling between an aspect and the base application (component).

3 ALICE

Alice is an aspect-oriented programming based approach that addresses the open issues by enabling (a) the modularization of infrastructural services and (b) by providing a clean programming model that (c) does not tie a component to a specific component framework. Nevertheless, a component will have a well defined component model.

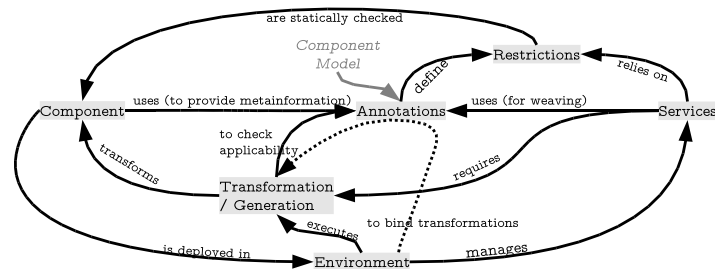


Figure 1: An overview of Alice

An overview of Alice visualizing the important parts and their relationships is shown in figure 1. As indicated by figure 1, a central feature of Alice is its use of standard Java 1.5 annotations [1] in both the implementation of the components as well as the implementation of the services. As far as the component programming model is concerned, the main purpose of annotations is to provide additional meta information about components they are associated with. By using annotations the component developer defines (a) the type of the component and its business methods as well as (b) how the component interacts with the environment; in other words a component's model is determined by the usage of specific annotations.

In its role of defining the type of a component, an annotation serves two purposes. First, it can define implementation restrictions that have to be followed by

the component developer and are statically checked at deployment time (e.g., synchronization primitives must not be used or a business method must be public) [5]. The second role of annotations is to specify requirements on the component environment. For example, the annotation `@BusinessMethod` in listing 1 only provides the meta information that the method implements a part of the business logic while the annotation `@SessionHandling` provides the meta information that we have a session component as well as requires that a service is available at runtime that provides session handling functionality.

```

1 @SessionHandling public class ShoppingCart{
2     @BusinessMethod public void addItem(Item item) { ... }
3     @BusinessMethod public void checkout() {...}
4 }

```

Listing 1: Excerpt of the implementation of a `ShoppingCart` component

Additional predefined annotations are available if a component must interact with a service. In addition, an annotation can be used by the services, as a means to identify relevant points, where to join the execution. Similar to AspectJ a service (aspect) in Alice defines pointcuts and advice. In addition, the developer of an infrastructural service in Alice can rely on the knowledge that annotations imply certain restrictions on the implementation of the components and uses the meta information encoded by annotations as a means to safely identify join points. In listing 2 an example pointcut and advice definition is shown. In this case the pointcut selects all methods that are annotated with the `BusinessMethod` annotation and for each join point selected by the pointcut the method `onExecution` is executed before. Passed to the method are the context information that are available at a join point, i.e. the receiver of the call or the current instance.

```

1 @Advice (pointcut="annotatedMethod('BusinessMethod')",type="before")
2 public void onExecution(Context context){ /* do something*/ }

```

Listing 2: An example Pointcut and Advice

So far the programming model for components has been discussed. With regard to programming the services, annotations serve two purposes: (a) to check the applicability of transformations, and (b) to identify join points. The environment has one part to handle the deployment of components and one part to handle the interaction between a component and the services. At deployment the component is verified to satisfy all implementation restrictions, that all necessary services are available, and that the transformations are executed. After that, the functionality of the environment is to enable the interaction between a component and a service.

4 FUTURE WORK & SUMMARY

The main goal of this work is to present a concise programming model that represents a significant improvement when compared with the current state-of-the-art in CBSD. It allows the separation of infrastructural services in off-the-shelf reusable aspects. This will be made possible by a set of annotations which are currently under



development and which are to be used by the component developer to provide additional information about the component and the join points for aspects. The aspect developer uses the annotations to bind the functionality to a component without requiring any knowledge about a components concrete implementation. Thus far, compilation and execution speed of the prototype under development is not targeted. Based on the specification of a set of service independent annotations, and the implementation of a small set of services which rely on those annotations, the approach will be assessed by developing a small demo application. Subject of the assessment is the off-the-shelf reusability of a service, i.e. whether it is possible to directly use a service or if adaption for a specific component or set of components is still required. Further, we will assess the approach by determining the achieved level of modularization between different services: two services will be considered well modularized if they can be developed independently of each other.

REFERENCES

- [1] J. Bloch. A Metadata Facility for the Java Programming Language. Java Specification Request 175, SUN Microsystems, 2002.
- [2] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE - Towards an Aspect Based, Programmable and Extensible Middleware Framework. In *Proceedings of ECOOP 2004*. Springer.
- [3] Adrian Colyer and Andrew Clement. Large-scale aop for middleware. In *Proceedings of AOSD 2004*. ACM Press.
- [4] Frédéric Duclos, Jacky Estublier, and Philippe Morat. Describing and using non functional aspects in component based applications. In *Proceedings of AOSD 2002*. ACM Press.
- [5] Michael Eichberg, Mira Mezini, Thorsten Schäfer, Claus Beringer, and Karl-Matthias Hamel. Enforcing system-wide properties. In *Proceedings of ASWEC 2004*. IEEE Computer Society.
- [6] Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-based Crosscuts. In *Proceedings of AOSD 2003*. ACM Press.
- [7] JBoss Inc. JBoss AOP Beta4. <http://www.jboss.org>, 2004.
- [8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP 2001*. Springer.
- [9] R. Laddad. *AspectJ in Action*. Manning, 2003.

- [10] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of ICSE 2000*. ACM Press.
- [11] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of AOSD 2003*. ACM Press.
- [12] Mati Shomrat and Amiram Yehudai. Obvious or not? regulating architectural decisions using aspect-oriented programming. In *Proceedings of AOSD 2002*. ACM Press.
- [13] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of OOPSLA 2002*. ACM Press.
- [14] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of AOSD 2003*. ACM Press.
- [15] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE 1999*. IEEE Computer Society Press.
- [16] Markus Völter, Alexander Schmid, and Eberhard Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. John Wiley & Sons, November 2002.
- [17] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of ICSE 1999*. IEEE Computer Society Press.
- [18] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14, November 2003.

ABOUT THE AUTHORS



Michael Eichberg is a PhD student and research assistant at the Software Technology Group at Darmstadt University of Technology, Germany. He can be reached at eichberg@informatik.tu-darmstadt.de. See also <http://www.st.informatik.tu-darmstadt.de/staff/Eichberg>.