Towards a symbiosis between Aspect-Oriented and Component-Based Software Development

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Viviane Jonckers
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 67
viviane@info.vub.ac.be

ABSTRACT

In this paper we present a novel approach, called FuseJ, for achieving a symbiosis between aspect-oriented and component-based software development. We build on previous research that proposes a new aspect-oriented programming language tailored for the component-based field, called JAsCo. Although JAsCo provides us with some nice results, we argue that a symbiosis between aspects and components is essential. To achieve this symbiosis, we describe the first steps towards a new component model, where both aspects and components are described in the same base component language. Each component is equipped with a number of homogeneous gates that allow accessing a particular feature. An application is assembled by interconnecting these gates, using explicit connectors, which contain the full expressive power for specifying crosscutting communication. As crosscutting behavior is specified as regular components, aspects and components can not be differentiated and a true symbiosis has been obtained.

Keywords: aspect-oriented software development - component-based software development - symbiosis - component model

1. INTRODUCTION

For a long time, object-oriented software development (OOSD) was considered the holy grail of software engineering. When an object-oriented application is built, it is split up in a set of classes which are able to perform one or more specific tasks for the system. Although OOSD considerably improved the development of software applications, it did not cure all problems experienced during the software engineering process. For some years now, component-based software development (CBSD) and more recently aspect-oriented software development (AOSD) have been proposed to tackle these problems.

One of the problems of OOSD, is the hard-coupled collaboration between the classes contained within the system. CBSD presents itself as a solution for overcoming this hard-coupling. In CBSD, full-fledged software systems are developed by assembling a set of pre-manufactured components. Each component is a black-box entity that can be deployed independently and is able to provide one or more specific services to the system [14]. The deployment of this paradigm drastically improves the speed of development. Also, the quality of the produced software is improved, as domain-specific components are reused several times.

AOSD [1,6] on the other hand, aims at improving the "separation of concerns"-principle in OOSD. When a software system is developed, properties of the application should ideally be described independently from each other. This paradigm makes it possible to independently analyze, reuse, change and extend the features provided by the system. OOSD tries to achieve this principle by providing a class-model in which the properties of a system can be described. Some properties of a software-system however, called aspects, can not be cleanly modularized using OOSD, as their implementation crosscuts several classes of the system. This is mainly caused by the tyranny of the dominant decomposition [11], as only one separation dimension is available for describing the properties of the system. As a result, the implementation of aspects is spread among several classes of the system. Examples of such aspects within the system are synchronization and logging. To solve this problem, AOSD proposes to describe each crosscutting aspect as a separate entity, which is weaved in the base implementation of the system later on. This way, other parts of the system are not affected when aspects are added, edited or removed.

Nowadays, several AOSD-technologies, such as AspectJ [2], HyperJ [10], and Compostion Filters [3], are available, for describing crosscutting aspects in the OOSD-context. Little by little, the possibilities of AOSD are researched in a component-based context. Similar to OOSD, aspects such as persistence and accounting are encountered, which crosscut several components from which the system is assembled. Consequently, the ideas behind AOSD should also be integrated into the CBSD-

context. The other way around, namely the integration of CBSD within the AOSD-context, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components: it should be fairly easy to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, makes aspects reusable and their deployment easy and flexible.

Combining the ideas behind AOSD and CBSD would consequently be a valuable contribution to both paradigms. The available AOSD and CBSD technologies however can not be straightforwardly integrated, because of several restrictions:

- Nearly all AOSD-approaches describe aspects with a specific context in mind, which limits reusability.
- The deployment of an aspect within a software-system is at the moment rather static, as aspects loose their identity when they are integrated within the base-implementation. As a result, aspects are not able to exhibit the same plug-and-play characteristic as components.
- The communication between components depends on the employed component model. Current AOSD-technologies however, are not suited to deal with these specific kinds of interactions.

JAsCo [5] is our first experiment to integrate the ideas behind AOSD and CBSD, as it allows describing reusable aspects which can deployed independently in a component-based context. JAsCo differentiates three kinds of entities: aspects, components and connectors, which are described making use of special, dedicated languages. Case-studies however, performed using JAsCo, illustrate that no real difference can be found between aspects and components. They are both reusable, independently deployable entities that deliver one or more specific services for the system, with the exception that their mutual communication is defined along another separation dimension [9].

In this paper, we present the first steps towards a symbiosis between aspect-oriented and component-based software development, by introducing a new component model, called FuseJ, where no distinction is made between aspects and components. Both are described in a base component language, and their collaboration is once more specified by making use of connectors. The next section describes JAsCo, our first experiment for integrating AOSD and CBSD. Section three presents some critical observations about JAsCo and illustrates the necessity for a symbiosis between aspects and components. Section 4 introduces the first concepts of the FuseJ component model. In section 5 we describe some related work and section 6 presents our future research. Finally we state our conclusions.

2. JASCO

JAsCo was our first experiment to achieve integration between aspect-oriented and component-based software development, by providing an aspect-oriented extension for the JavaBeans component-model. JAsCo is primarily based upon two existing AOSD approaches: AspectJ [2] and Aspectual Components [7]. AspectJ's main advantage is the expressiveness of its "join point"-language, as it allows describing properties of a system that interact on very specific points in the execution of the application. However, aspects described making use of AspectJ, are not reusable, as the context on which an aspect needs to be deployed is specified directly in the aspect-definition. To overcome this problem, Karl Lieberherr et al introduce the concept of Aspectual Components. They claim that doing aspect-oriented programming means being able to express each aspect separately, in terms of its own modular structure. Using this model, an aspect is described as a set of abstract join points which are resolved when an aspect is combined with the base-modules of a software system. This way, the aspect-behavior is kept separate from the base components, even at run-time. JAsCo combines the expressive power of AspectJ with the aspect independency idea of Aspectual Components. To achieve this objective, JAsCo introduces two new entities: *aspect beans* and *connectors*. An aspect bean is an extension of a Java Bean component, which is able to specify crosscutting behavior. A connector on the other hand is responsible for deploying the crosscutting behavior of the aspect beans into a specific context and for declaring how several aspects collaborate.

In this section, we present the basic features of the JAsCo-language. For more information about this approach, and how its underlying component model is implemented, we refer to [13].

2.1 THE JASCO LANGUAGE

The JAsCo language stays as close as possible to the regular Java syntax and constructs and introduces two new concepts: *aspect beans* and *connectors*. Aspect beans are used for describing some functionality that would normally crosscut several components from which the system is composed. An example of such crosscutting concerns is *caching*. Some features of a system consume a lot of resources to accomplish their task. Caching some of their resulting output could drastically improve the performance of the system. This caching-aspect could be a valuable property for an online booking/searching system for hotels. Instead of executing the search-query in the database for each request of a customer, a set of query-results could be cached to improve the performance of the system. Figure

1 illustrates the implementation of this caching-aspect in JAsCo.

```
1  class CachingManager {
2
3    Cache cache = new Cache();
4    void setRecyclingRate(int sec) {
5      cache.recylingRate(sec); }
6
7    hook CacheControl {
8                                                    When?
9      CacheControl(method(..args)) {
10       execute(method); }
11                                                   What?
12     replace() {
13       if(cache.cached(method,args) {
14         return
15          cache.getCached(method,args); }
16       else {
17         Object re = method(method,args);
18         cache.cache(method,args,result);
19         return re;
20       }
21     }
22   }
23 }
```

Figure 1: The JAsCo-aspect for caching.

Aspect beans usually contain one or more hook-definitions (line 7 till 22), and are able to include any number of ordinary Java class-members (line 3 till 5), which are shared amongst all hooks of the aspect. A hook is used for defining when the normal execution of a method should be cut, and what extra behavior there should be executed at that precise moment in time. For defining when the behavior of hook should be executed, each hook is equipped with at least one constructor (line 9 till 10) that takes one or more abstract method parameters as input. These abstract method parameters are used for describing the context of a hook. The *CacheControl*-hook specifies that it can be deployed on every method that takes zero or more arguments as input. The constructor-body defines how the join points of a hook initialization are computed. In this particular case, the constructor-body (line 10) specifies that the behavior of the *CacheControl*-hook should be executed whenever method is executed. The behavior methods of a hook are used for specifying the various actions a hook needs to perform whenever one of its calculated join points is encountered. Three kinds of behavior methods are available: *before*, *after* and *replace*. The *CacheControl*-hook specifies only one behavior method (line 12 till 20). The replace behavior method specifies that the cache should be checked if it contains the output-value for the specific input-values of the arguments of a method. If so, the cached result is returned. In the other case, method is executed, and its result is cached for later use. Note however, that the Cache-object will recycle its content depending on the number of seconds that are specified.

Connectors are used for initializing a hook with a specific context (methods or events). A hook initialization takes one or more method or event signatures as input. Figure 2 illustrates the *CachingConnector*. This connector initializes a *CachingControl*-hook with the *getHotels*-method of the *BookHotel*-component (line 3 till 6). After initializing this hook, the *CachingConnector* specifies the execution of the replace behavior method (line 9) and sets the cache recycling rate on 60 seconds (line 8). Consequently, the *CachingConnector* has following implication: check if some cached result exists whenever a customer requests the available hotels for a specific city. If so, return the cached lists of hotels. Otherwise, execute the query and cache its result.

```
1  static connector CachingConnector {
2                                                    Where?
3    CachingManager.CacheControl ca =
4      new CachingManager.CacheControl (
5        List BookHotel.getHotels(String)
6    );
7
8    ca.setRecylingRate(60);
9    ca.replace();
10
11 }
```

Figure 2: The JAsCo-connector for caching result of the HotelBook-component.

3. ASPECT/COMPONENT SYMBIOSIS

Several case-studies have been performed, making use of the JAsCo aspect-oriented component language. JAsCo has been integrated into the visual component composition environment PacoSuite [15], for implementing invasive composition adapters. Research has been conducted to represent business-rules as JAsCo-aspects to incorporate them in a software-system. In the future, research is considered in the webservice-domain and a large-scale project is planned to increase the maturity of JAsCo.

Although the JAsCo-approach is a valuable contribution to research that tries to achieve integration between AOSD and CBSD, some criticism is required. The JAsCo-model makes a distinction between three kinds of concepts: components, aspects and connectors. Although components and aspects are described by making use of special dedicated languages, no fundamental difference can be found between both entities when aspects are compared to the ECOOP '96 definition [14] of components: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*".

When comparing aspects to this definition, the following observation can be made: Similar to components,

- aspects provide some functionality for the system by making use of contractually specified interfaces.
- aspects need to be described independently from a specific context.

- aspects can be combined with other components or aspects, to become useful to the system.

This observation has two possible consequences: either the definition of a component requires updating or an aspect should be considered as a regular component. In our opinion, the second consequence is the more likely one. Also, it should be possible to reuse components within a variety of applications. For some functionalities of a system however, it is very difficult to decide beforehand whether it should be specified as an aspect or as a component. Encryption for instance, could be crosscutting in some applications and non-crosscutting in others. As a result, an aspect-oriented version of the encryption-functionality can no longer be used in a non-crosscutting way, which limits its reusability. When aspect-oriented and regular components are however expressed uniformly, this problem does no longer exist.
An aspect can thus be conceived as a regular component, with the exception that its communication with other components is defined along another separation dimension [9]. This conclusion is also noticeable in the JAsCo aspect-language. Only one additional construct is provided, on top of the regular JAVA-constructs, which is used to describe the context of an aspect bean. The remaining parts of the aspects are described by making use of regular JAVA.

One way to achieve a kind of symbiosis between aspects and components in the JAsCo language is by describing the aspect-behavior (the JAVA-part of the aspect) in a separate component. When this principle is applied onto the caching example introduced in section 2, it can be rewritten as illustrated in figure 3.

```
1  class CachingManager {
2
3     CacheComp cache = new CacheComp();
4     void setRecyclingRate(int sec) {
5       cache.recylingRate(sec); }
6
7     hook CacheControl {
8
9       CacheControl(method(..args)) {
10        execute(method); }
11
12      replace() {
13        cache.execBehavior(method,args); }
14    }
15 }
```

Figure 3: New JAsCo-aspect for caching.

The behavior of the Caching-aspect (the replace behavior of the *CacheControl*-hook in figure 2) is now specified in a separate component called *CacheComp*. The *CachingManager* aspect itself only contains the specification of the abstract aspect context (line 9 till 10) and the execution of the caching-behavior (line 12 till 13). For deploying the new implementation of the *CachingManager*-aspect within the system, a connector is again required. The implementation of this connector is similar to the one of figure 2.

Although some kind of symbiosis is achieved by describing aspects in this manner, three critical observations can be made on this approach. First of all, the *CachingManager*-aspect is not an aspect-bean anymore. It only specifies an abstract context and the execution of some method when the concrete context is encountered. As it does no longer implement any particular behavior, we can no longer consider it an aspect-bean. Secondly, the behavior of the caching-aspect is now scattered amongst two places: the *CacheComp*-component and the *CachingManager*-aspect. Although the aspect-behavior is removed from the aspect itself, an explicit call to the component that implements the behavior is still required. In fact, we could even argue that the implementation of the *CacheComp*-component crosscuts the *CacheControl*-hook, as this hook is only responsible for catching a specific point in the execution of the application. Thirdly, the *CachingManager*-aspect becomes quite obsolete when aspects are implemented this way, as for applying this aspect within the system, a connector is still required. In fact, the aspect-definition of figure 3 only specifies an extra level of indirection, which is actually not required. The aspect-bean used to describe the behavior of the caching-aspect. Now however, it is reduced to a kind of template for the abstract application of an aspect.

When comparing aspects to the ECOOP-definition of components, it is clear, that aspects and component are quite similar. From the arguments above, it is proved that JAsCo is not able to enforce symbiosis between both entities. The next section describes our first steps towards a new component-model where we want to accomplish symbiosis between aspects and components.

4. FUSEJ COMPONENT MODEL

To accomplish symbiosis between aspects and components, we propose a new component model, FuseJ, which makes no distinction between aspects and components at both implementation and assembly time. The component model features three layers: a component layer, a gate layer and a communication layer. Figure 4 illustrates this new component model with the hotel booking system introduced in section two as a concrete example.

The aspects and the base components of a system are all part of the component layer. The functionality of both these types of entities is described in a base component language, and no specific language-features are provided for specifying aspects. As a result, there is no way to distinguish an aspect from a component when observing their implementation. All components contained within the component layer are black-box and completely independent of each other. Consequently, no hard

linking between components exists. In our online hotel booking system, two components are available: the booking component and the caching component.

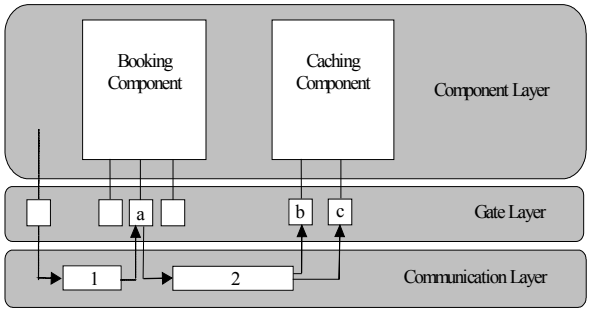

Figure 4: FuseJ component model

The services provided by a component can not be accessed directly. All communication with or from a component needs to pass through the gate layer. Each component within the system is provided with a number of gates that offers access to features provided by the component. A gate can thus be observed as some kind of guardian of a two-way channel that allows accessing the internals of a component. It is the responsibility of the component implementer to provide each component with at least one gate. The caching component for instance, provides two gates. Gate *c* allows access to the component-feature that allows setting the caching recycle-rate. A request to this gate will be translated into a call to the *recylingRate*-method of the cache-component. Gate *b* allows accessing the caching-feature of the caching-component, previously implemented as an aspect. A request to this gate will be translated into a crosscutting execution of the method that implements this caching behavior. The mapping of a gate onto a component is not always supposed to be a one-on-one method mapping. Several method calls could be required to be able to perform a specific feature provided by the gate. This control-flow within the component is however transparent to the user. The nice concept about gates is that all gates are homogenous to the component composer, in the sense that it does not matter if a gate provides access to a feature that implements some aspect-behavior or a feature that implements some base-functionality for the system. As already mentioned, gates are two-way channels. Incoming communication has following semantic: "Execute the feature of the component the gate provides access to." Outgoing communication has following semantic: "Whenever the feature of the component the gate provides access to, is executed, do something else." "Something else" depends on the feature of some other component the outgoing communication is referring to.

The communication between gates is specified by making use of connectors, situated in the communication layer. A connector is a one-way channel for interrelating the various features of the components. A connector is thus responsible for combining the outgoing communication of a component with the incoming communication of another component. In case of the hotel booking example of figure 4, connector 1 specifies that some component of the system queries the booking-component (gate a). Connector 2 specifies that whenever this query (gate a) is executed, the caching feature of the caching component is executed (gate b) and the recycling rate (gate c) is set. Take in mind that connectors are n-ary entities. As a result they are able to contain multiple inputs and outputs.

When comparing the FuseJ component model to other technologies that allow describing crosscutting behavior, an evolution can be observed. In AspectJ, the expressive power for specifying crosscutting behavior is completely contained within an aspect, as it describes both the behavior and the concrete linking points with the base-application. JAsCo breaks this crosscutting specification up into an aspect that describes the behavior and the abstract context, and a connector which specifies the concrete context. FuseJ is the next step in this evolution. No real aspect definition is found anymore, because aspects are defined as regular components. The full crosscutting power is now contained solely within the connectors. As a result, the power of the FuseJ component-model is dependent on the expressive power of the connector specifications, as these have to allow both regular and crosscutting communication at the same time.

5. RELATED WORK

Jiazzi [8] is another approach that combines aspect-oriented ideas with component based software development. In Jiazzi, software is composed of different units that can be compiled separately. Similar to FuseJ, units do not need to declare whether their behavior is crosscutting. Units themselves are able to export signatures. A signature is similar to a gate because they both specify an interface to a component. Gates however can also specify all possible join points explicitly. A separate linking language, similar to the FuseJ connector language, is used to specify the interactions between the reusable units. The FuseJ connector language extends the linking language of Jiazzi as it allows specifying more complex combinations of components.

Some other approaches also force a component developer to explicitly specify the possible join points of the component, like for example AspectLagoona [4] and µ-Dyner [12].

6. FUTURE RESEARCH

This paper only present the first steps towards a new component model, called FuseJ, where no distinction is made between aspects and components. In the future, the

concepts and ideas that were presented need to be elaborated further on. In particular, the mapping of the gates onto the components and the various communication mechanism provided by the connectors need to be investigated. This research will be performed iteratively, making use of a set of case-studies.

7. CONCLUSIONS

Current AOSD-technologies consider aspects and components to be two separate entities, as both are described in their own dedicated language. However, when comparing aspects to the ECOOP-definition of components, no real difference can be found between both entities. Therefore, we propose a new component model, where no distinction is made between aspects and components. Both are described in some base component language, and no special language features are provided for specifying aspects. Access to the features provided by a component is supplied by means of gates. These gates are homogenous in the sense that it does not matter if a gate provides access to a feature that implements some crosscutting behavior or a feature that implements some base-functionality of the system. To interconnect the various components, connectors are used. As crosscutting communications have been entirely moved to the connector-layer, the usefulness of this model is dependent on the expressive power of the connectors.

The FuseJ component model has some promising advantages. No distinction between the crosscutting and non-crosscutting behavior is made anymore, as components are expressed in terms of features which do not imply a dimension. As a consequence, a component developer does not need to choose at component development time whether his component describes crosscutting or regular behavior. This increases reusability of the developed components. Another advantage of this approach is that because of the feature-interface of the gates, the interior of a component is not revealed at all. This facilitates to replace or update a component, as long as the new component still complies with the old feature-interface. Also, the component-model is hierarchical, as several assembled components can again be used as a single component, ready for composition. A disadvantage of this feature-concept is that join points are specified on a higher level of granularity than those found in most aspect-oriented technologies.

8. ACKNOWLEDGMENTS

We owe our gratitude to Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Since October 2000, Wim Vanderperren is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: “Fonds voor Wetenschappelijk Onderzoek”).

REFERENCES

[1] AOSD Website: http://www.aosd.net.

[2] AspectJ Website: http://www.aspectj.org.

[3] Bergmans, L. and Aksit, M. **Composing Crosscutting Concerns Using Composition Filters.** Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[4] Gal, A., Franz, M. and Beuche, D. **Learning from Components: Fitting AOP for System Software.** In proceedings of ACP4IS workshop at AOSD 2003, Boston USA, March 2003.

[5] JAsCo Website: http://ssel.vub.ac.be/jasco

[6] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A. and Murphy, A. **Aspect-Oriented Programming.** In proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.

[7] Lieberherr, K., Lorenz, D. And Mezini, M. **Programming with Aspectual Components.** Technical Report, NU-CSS-99-01, March 1999.

[8] McDirmid, S. and Hsieh, W.C. **Aspect Oriented Programming with Jiazzi.** In Proceedings of AOSD International Conference, Boston USA. ACM Press. March 2003.

[9] Ossher, H., and Tarr, S. **Multi-Dimensional Separation of Concerns in Hyperspace.** Position paper at the ECOOP ’99 Workshop on Aspect-Oriented Programming, Lisbon Portugal. June 1999.

[10] Ossher, H. and Tarr, P. **Multi-Dimensional Separation of Concerns and The Hyperspace Approach.** In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

[11] Parnas, D. L. **On the Criteria to be Used in Decomposing Systems into Modules.** In Communications of the ACM. Vol. 15. No. 12. Pages 1053-1058. December,1972.

[12] Ségura-Devillechaise, M., Menaud, J. and Muller, G. **Web Cache Prefetching as an Aspect: Towards a Dynamic Weaving Based Solution.** In Proceedings of AOSD International Conference, Boston USA. ACM Press. March 2003.

[13] Suvée, D., Vanderperren, W., and Jonckers, V. **JAsCo: an Aspect-Oriented approach tailored for CBSD.** In Proceedings of AOSD International Conference, Boston USA. ACM Press. March 2003.

[14] Szyperski, C. **Component software: Beyond Object-oriented programming.** Addison-Wesley, 1998.

[15] Vanderperren, W., Suvee, D. and Jonckers, V. **Invasive Composition Adapters: an aspect-oriented approach for visual component-based development.** In proceedings of ACP4IS workshop at AOSD 2003, Boston USA, March 2003.