FuseJ: Achieving a Symbiosis between Aspects and Components

Davy Suvée
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, 1050 Brussels, Belgium
dsuvee@vub.ac.be
http://ssel.vub.ac.be/Members/dsuvee

Keywords: component-based software development, aspect-oriented software development, symbiosis.
Classification: 5 months in first year's PhD work.

1 Introduction

For a long time, object-oriented software development (OOSD) was considered the holy grail of software engineering. 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 some of these problems.
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 that is able to provide one or more specific services for the system [4]. AOSD on the other hand, aims at improving the "separation of concerns" principle. Some properties of an application can not be nicely modularized using current software engineering methodologies, as they crosscut several entities of the system. AOSD aims at solving this problem by introducing a new concept, called an aspect, which enables the modularization of crosscutting concerns [1].
Nowadays, several AOSD technologies are available for describing crosscutting concerns in an object-oriented context. Little by little, the possibilities of AOSD are explored in a component-based context. Similar to OOSD, CBSD suffers from the tyranny of the dominant decomposition [2]. As a result, integrating the ideas behind AOSD into CBSD is required. The other way around, namely the integration of the ideas behind CBSD into AOSD, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components, and introducing a similar concept in AOSD, makes aspects reusable and their deployment easy and flexible.
In this paper, we present the first steps towards a new component model, called FuseJ, which aims at achieving a symbiosis between AOSD and CBSD, by unifying aspects and components. The next section introduces the goal of this research by sketching the shortcomings of our previous research which aimed at integrating the ideas behind AOSD and CBSD [3]. Section three introduces the first concepts of the FuseJ

component model. Section four discusses the current status of the FuseJ research. Finally, we state our conclusions.

2 Research Goal

As already mentioned, integrating the ideas behind AOSD and CBSD would be a valuable contribution to both paradigms. The JAsCo-language [3] was our first step towards this integration. JAsCo is an aspect-oriented extension for the Java Beans component model which allows describing reusable aspects, independently from a specific context. JAsCo differentiates three kinds of entities: aspects beans, components and connectors, which are described making use of dedicated language elements. An aspect bean is a regular Java Bean, extended with a number of hooks, for describing crosscutting behavior. Connectors on the other hand, are responsible for deploying one or more aspect beans within a specific component context. This way, whenever the context of an aspect bean is executed, the crosscutting behavior, defined in the aspect bean, is performed. For more information on the JAsCo approach, we refer to [3].

Several case-studies have been performed, making use of the JAsCo aspect-oriented component language. Although the JAsCo approach is a valuable contribution to research that aims at achieving integration between AOSD and CBSD, some criticism is required. Similar to classical AOSD-approaches, an aspect bean is responsible for specifying two things: *behavior logic* and *a description on how the aspect bean crosscuts the base implementation of the system*. But consider for instance the need for a service which supplies encryption. In some applications, encryption is implemented as a regular component, and in others, it is implemented as an aspect. The implementation of the encryption logic itself however, remains the same in both cases. The only difference between the aspect version of the encryption functionality and its component version is the way they interact with the rest of the application.

The goal of our research is to achieve symbiosis between AOSD and CBSD. In this research, no distinction is made any longer between aspects and components. Aspects are implemented as regular components, omitting a description of how they crosscut the base implementation of the system. The crosscutting interaction is consequently specified on a different level.

3 FuseJ Component Model

To achieve symbiosis between aspects and components, we propose a new component model, called FuseJ, which makes no distinction between aspects and components at implementation time, at assembly time and at run-time. The FuseJ component model contains three layers: a *component layer*, a *gate layer* and a *communication layer*. Figure 1 illustrates the FuseJ component model, making use of a *gameserver application* as a basic example. Several players are able to connect to the server for playing a game and the gameserver application itself uses logging for statistical purposes.

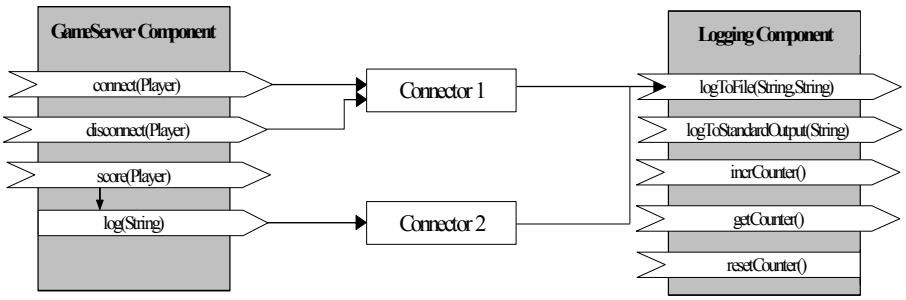

Figure 1: FuseJ Component Model

All aspects and base components of a system are part of the *component layer*. The behavior of both these types of entities is described in some base component language, and no specific language elements are provided for specifying aspects. As a result, there is no way to distinguish aspects from components when observing only their implementation. Each component contained within the component layer is a black-box entity and is specified independently of other components. In our example, the gameserver application contains two components: a *GameServer* component and a *Logging* component.

Each component contained within *the component layer* provides a number of services, which we call *features*. These features however, can not be accessed directly. All communication with or from a component needs to pass through *gates*, which are located in *the gate layer*. A gate is the only entrance point of a component, and provides access to some feature the component provides. As a result, a gate can be conceived as some kind of guardian of the internals of a component. A request to a gate is mapped onto the execution of one or more methods, contained within the component. This control-flow is however transparent to the user. It is the task of the component implementer to provide each component with a number of gates and to supply the corresponding method mappings. The *Logging* component for instance, supplies four gates. Some of these gates, such as *getCounter*, provide access to features which could be part of a regular component. Other gates, such as *logToStandardOutput* and *logToFile,* provide access to features which are typical examples of crosscutting behavior. In an aspect-oriented context, these features of the *Logging* component would typically be implemented as an aspect. The nice concept about gates however, is that they are homogenous to the component composer: it does not matter if a gate provides access to a crosscutting feature or a non-crosscutting feature, as this is not defined in beforehand, nor in the gate, nor in the component itself.

Gates are two-way channels. Incoming communication has the following semantics: "Execute the feature of the component the gate provides access to". Outgoing communication on the other hand, has the following semantics: "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. Whenever the *connect* feature of the *GameServer* component is for instance requested, its gate commands the execution of the feature(s)

of the gate(s) it is connected to. It is possible to disallow the incoming communication of a gate. This means that the component does not provide an implementation for this feature, but that its corresponding gate needs to be connected to the feature of another component to perform its functionality. An example of this case is the *logging* feature of the *GameServer* component, which is used by the *scores* feature for keeping statistics of the players' scores. It is also possible to disallow the outgoing communication of a gate. This is for instance the case for the *resetCounter* feature. Blocking outgoing communication could be useful in cases were no interference with a feature should be allowed, this for instance because of performance issues.
The communication between gates is specified by making use of connectors, situated in the *communication layer*. A connector is responsible for specifying the communication between the gate(s) of one component with the gate(s) of another (or the same) component. The gameserver application illustrated in figure 1 displays two interactions between the *GameServer* component and the *Logging* component. The implementer of the *GameServer* component anticipated the requirement that whenever a player scores a point, this event should be logged for statistical analysis. *Connector 2* is used to enable this interaction. It connects the gate of the *logging* feature of the *GameServer* component with the gate of the *logToFile* feature of the *Logging* component. This kind of connector is quite similar to the connectors found in most component-based models. Imagine now that our statistical analysis also wants to include the scoring rate per hour for each player. For enabling this property, it should be possible to also log the connection time of each user. The implementer of the *GameServer* component did however not anticipate this kind of logging. However, this interaction can still be enabled, by connecting the gates of the *connect* and *disconnect* feature of the *GameServer* component with the gate of the *logToFile* feature of the *Logging* component. Observe that this kind of communication between components is defined as being crosscutting. It is however the connector that is responsible for implementing this crosscutting interaction, as the *Logging* component can be used as both crosscutting and non-crosscutting at the same time. Also, take in mind that connectors are n-ary entities, i.e. they can contain multiple inputs and outputs.

4 FuseJ Status

We are experimenting with a prototype Java implementation of the FuseJ component model to demonstrate the effectiveness of our three layer model. Gates are implemented manually making use of a simple *event-listener* protocol. Connectors are also implemented manually and specify the interaction between gates. In the future however, a big part of the implementation of gates and connectors could be generated automatically. We are also considering two approaches for describing connectors: a dedicated language, or an implementation in regular Java. A dedicated language eases the development of connectors, as special language constructs are available for expressing specific interactions. An implementation in regular Java

however, makes it possible to provide connectors with gates, which would again provide a means for enabling (crosscutting) interactions on the connectors themselves.

5 Conclusions

Current AOSD technologies consider aspects and components to be two separate entities as both are described by their own dedicated language elements. Our previous research however indicates that aspects can be considered as regular components. The only difference between both entities is that their mutual interaction is defined differently. Therefore, we propose a new component model, called FuseJ, where no distinction is made between aspects and components. Components are expressed in terms of features, which are accessed making use of homogeneous gates. These gates are then combined by means of connectors, which are used to describe the (crosscutting) interactions between the various features provided by the components. The FuseJ component model has some promising advantages. The reusability of components is increased, as a component developer does not need to decide at development time whether his component describes (non-)crosscutting behavior. As the interior of a component is not revealed, it is easy to replace or to update a component, as long as it complies with the old feature interface. The FuseJ component-model is also hierarchical, as several assembled components can again be used as a single component, ready for composition. Also, as crosscutting features are implemented within regular components, it is possible to stack crosscutting features on top of each other. A disadvantage of gates however, is that join points are specified on a higher level of granularity than those found in most aspect-oriented technologies.
This paper only presents the first steps towards the new FuseJ component model. In the future, the concepts and ideas that are presented here need to be elaborated further on. In particular, the various communication mechanism provided by the connectors need to be investigated, as the power of the FuseJ component-model is dependent on the expressive power of its connectors.

References

[1] 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, May 1997.

[2] 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.

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

[4] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.