

# Applying aspect-oriented programming ideas in a component based context: Composition Adapters.

**Wim Vanderperren**

System and Software Engineering Lab (SSEL)  
Vrije Universiteit Brussel (VUB)  
Pleinlaan 2, 1050 Brussels, Belgium  
wvdperre@vub.ac.be  
<http://ssel.vub.ac.be/members/wvdperre>

**Keywords:** component, composition, aspect-oriented programming.

**Classification:** 1 year's PhD work

## 1 Introduction

Aspect-oriented programming (AOP) is a new programming methodology that enables the modularization of crosscutting concerns. Until now, the emphasis of AOP research lays on being able to modularize these concerns in an object-oriented context. However, the same problem also applies to the component based software engineering domain. In this paper we propose a solution, namely composition adapters. We are able to cleanly modularize crosscutting concerns using composition adapters. Additionally, we develop algorithms to automatically weave these concerns into the composition. The next section describes the context in which this research is conducted, followed by an explanation of the research goal. Afterwards, different approaches are presented and our approach is described in more detail.

## 2 Research Context

Component Based Development is one of the research topics of the System And Software Engineering Lab. The research of the lab mainly focuses on lifting the abstraction level for the development of component-based systems. Until now designing and developing component-based applications turns out to be very hard. There is no support to check whether components are able to work together and the glue-code to make the components work together has to be written manually. Much of the existing glue-code in current systems is written to "hack" components together instead of following a careful design.

Explicit composition patterns that describe the interaction between a number of roles are introduced. A special kind of Message Sequence Charts (MSC) [1] is used as notation for the composition patterns. Components are also documented using these MSC's. Figure 1 and Figure 2 illustrate examples of respectively a component and composition pattern documentation. At composition time, each role of a composition pattern is filled in by a component. Automatic compatibility checks reject inappropriate components based on their documentation using automata theory. When

all the roles are filled, glue-code is automatically generated. This glue-code translates syntactical incompatibilities between the components and constrains the components their behavior as prescribed by the composition pattern. Additionally, a tool that realizes these ideas in a visual way has been implemented. Figure 3 shows a screenshot of this tool. For more information about the algorithms for compatibility checking and glue-code generation and about our approach on component-based development in general, see [2][3].

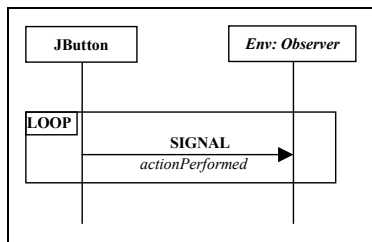


Figure 1: Usage Scenario of JButton bean.

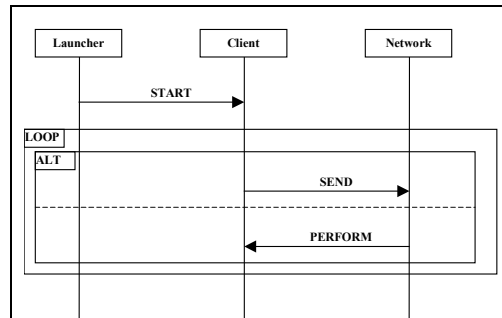


Figure 2: Example of a composition pattern.

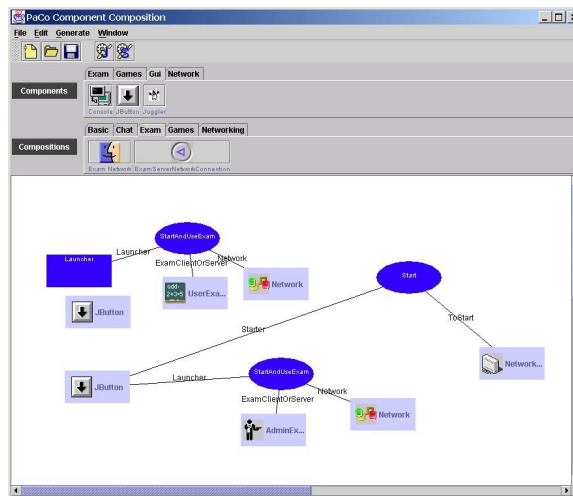


Figure 3: PacoSuite: our visual component composition prototype tool.

### 3 Research goal

A couple of case studies using the PacoSuite tool are done to validate our approach. Although these case studies were successful, we felt that some concerns of the application were not cleanly modularized. For example, one of the case studies is a distributed exam service. To introduce accounting in the exam service, all composition patterns have to be altered in the same way. Because we have no way to

describe these adaptations in a separate module, new composition patterns that include both the original and the accounting behavior have to be created. Another approach to introduce accounting consists of modifying existing components so that they are able to send accounting information to interested accounting components. The goal of my research is to be able to cleanly modularize crosscutting concerns in component-based systems. In other words, we try to recuperate the ideas from aspect-oriented programming into the component-based development domain. However, we do not want to lower the abstraction level. So, the formalism we introduce to capture crosscutting concerns has to allow component composition in a visual manner without in-depth technical knowledge of the components.

## 4 Possible Approaches

We see two different possibilities to modularize crosscutting concerns in our component-based context. The first solution consists of using a new component model that allows a component to describe adaptations in other components. Prof Lieberherr and others present a concrete proposal for such a component [4]. They call these components aspectual components. They propose to have a new type of interface that allows components to describe adaptations independent of the concrete components that will be adapted. At composition time, special compositions connect the adaptations with the concrete components. The adaptations are then weaved into the components using binary code adaptation. This approach is very powerful, because the adaptations are described by a programming language (in fact a special version of JAVA). Although this is an interesting approach, it is impossible to directly recuperate it in our component-based context. Because we do not want to lower the abstraction level, we have to come up with a (preferable graphical) notation of what the consequence of the adaptations on the exterior behavior of the altered components will be. This extra information is needed to allow automatic compatibility checking and glue-code generation.

Therefore, we propose to use another alternative, namely having special compositions that could adapt other compositions. This approach is clearly less powerful, but by far a more easier and flexible solution. Composition adapters are only able to alter the exterior behavior of components by re-routing or ignoring their messages. However, the code for the compositions is not yet generated, so adapting these compositions requires no code adaptation whatsoever.

## 5 Composition Adapters

We propose to document composition adapters by MSC's similar to regular composition patterns. Composition adapters consist of two parts, a context and an adapter part. The context part describes the behavior that will be adapted. The adapter part describes the adaptation itself. Figure 4 illustrates an example of a composition adapter. In this example, the composition adapter will re-route every occurrence of a SEND from role *Source* to role *Dest* through a *Logger* role. Suppose we apply this composition adapter to the composition pattern of Figure 2. Then we manually map

the *Source* role of the composition adapter onto the *Client* role of the composition pattern in Figure 2. Likewise, the *Dest* role is mapped onto the *Network* role. The result of applying the composition adapter is that every SEND from *Client* to *Network* will be sent through the *Logger* role (see Figure 5). The *Logger* role and the combined *Source/Client* and *Dest/Network* roles are afterwards filled in by concrete components. In the aspectual component approach, the *Logger* component would be an aspectual component that adds logging logic either to the component mapped on the *Source* role or the component mapped on the *Dest* role. Notice that from this example it seems useful to be able to express wildcard roles in composition adapters. Wildcard roles would be automatically mapped onto roles of the affected composition. This would free the component composer of manually mapping composition adapter roles.

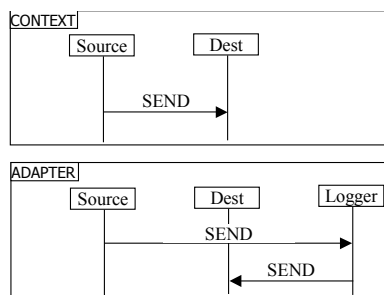


Figure 4: Logging composition adapter.

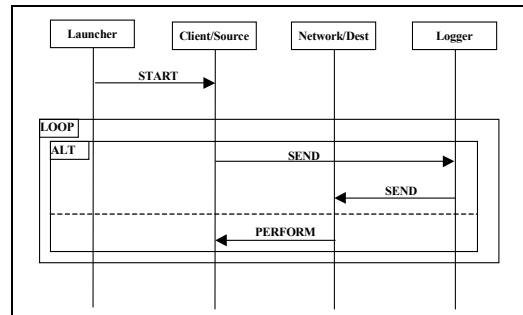


Figure 5: Logging composition adapter applied to the composition pattern of Figure 2.

## 6 Weaving a composition adapter into the composition

Automatically applying a composition adapter requires two steps. In the first step we check whether the adaptation makes sense, this means checking if the context of the composition adapter appears in the composition pattern the composition adapter is applied upon. Although this seems obvious from the example in Figure 4, where we just have to search for a SEND in the composition pattern of Figure 2, in most cases syntactically scanning the affected composition won't work. If the context is described by loops and/or other control blocks, a more evolved algorithm that matches the MSC's on a semantic level is needed. The algorithm we developed is based on automata theory. Due to space constraints the algorithm is only shortly sketched. Both the context of the composition adapter as the affected composition pattern are translated to a Deterministic Finite Automaton (DFA). Then, for each state in the DFA of the composition pattern the product automaton with the DFA of the adapter context is calculated. If one of these product automata is not empty, meaning that there exists at least one path from start to stop state, then it is possible to apply the adaptation there. Although this check may seem superfluous because not being able to apply the composition adapter doesn't do any harm, the result of the algorithm

is needed for the next step. Additionally, if a component composer applies a composition adapter, it is probably not her/his intention that this has no effect. Subsequently, the composition adapter has to be weaved into the composition. This algorithm is again based on automata theory. The result of the previous algorithm, namely the states where it is possible to apply the composition adapter, is needed here. The algorithm to weave in the composition adapter is rather complex and therefore not elucidated here due to space constraints. The general idea is to replace all paths where the context of the composition adapter applies with the adaptation. The automaton generated by the algorithm is used to check compatibility with filled-in components and to generate glue-code. For this we use our algorithms developed in earlier work [2][3].

## 7 Conclusions

In this paper, we build on our previous work to lift the abstraction level of component-based development. We notice that there are concerns that crosscut the component-based design and provide a solution. Using composition adapters, we are able to cleanly modularize crosscutting concerns. Furthermore, we develop algorithms to automatically weave these concerns into the composition.

Benefits do not only arise in the development of the component-based application, but also in evolutionary changes to the application. New requirements often lead to concerns that crosscut the existing component composition. Instead of having to re-wire all the components, the existing compositions are altered using composition adapters.

## References

- [1] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1993.
- [2] Vanderperren, W. and Wydaeghe, B. *Towards a New Component Composition Process*. In Proceedings of ECBS 2001, Washington., USA, April 2001.
- [3] Wydaeghe, B. and Vandeperren, W. *Visual Component Composition Using Composition Patterns*. In Proceedings of Tools 2001, Santa Barbara, USA, July 2001.
- [4] Lieberherr, K., Lorenz, D. and Mezini, M. *Programming with Aspectual Components*. Technical Report, NU-CCS-99-01, March 1999. Available at: <http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html>.