A pattern based approach to separate tangled concerns in component based development

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

ABSTRACT

This work builds on aspect-oriented software development ideas and our previous research where we lift the abstraction level of visual component based development. In component based development, the components are the natural unit of modularization. However, there will always be concerns that cannot be confined to one single component. We introduce composition adapters as a means to modularize crosscutting concerns in separate and reusable entities. A composition adapter describes an adaptation of the interaction protocol between a set of components. An important feature of a composition adapter is that the adaptations are described independent of a concrete API, making them highly reusable. Using composition adapters, we are able to weave crosscutting aspects in a component based application. The weaving algorithm uses automata theory to allow the state-based insertion of a composition adapter into the interaction protocol. This allows a seamless integration with our component based methodology. We embedded composition adapters and our algorithms into PacoSuite, a visual component composition tool that is used in our lab as a research vehicle. PacoSuite hides the underlying complexity to the component composer, rendering an easy to use visual component based development environment that includes now aspect separation features through composition adapters.

1. INTRODUCTION

Component based software development is considered a promising paradigm for curing the so-called software crisis [1]. The idea is that applications are created by composing reusable components. Hence both the software quality and the development speed improve substantially. At the System and Software Engineering Lab (SSEL) we have been doing research on a novel component based software development methodology for a couple of years. The major goal of our approach is to lift the abstraction level for component based software development. The success of design patterns [2] indicates that there exist collaboration patterns that are used frequently. Therefore, we introduce explicit and reusable composition patterns. A composition pattern is an abstract specification of an interaction between a number of roles. Our approach allows us to automatically verify whether a component is able to work as a role of a composition pattern prescribes. Moreover, we are able to generate glue-code that translates syntactical compatibilities between a number of components mostly automatically.

Another research direction that has received lots of attention in the last years is Aspect-Oriented Software Development (AOSD) [3]. Some aspects of an application cannot be cleanly modularized using current software engineering methodologies. Typical examples include logging or synchronization. The focus of AOSD research has been on separating crosscutting concerns in an object-oriented context. However, the same problem also applies to component based software development. To be able to separate crosscutting concerns in our component based context, we introduce the concept of a composition adapter. A composition adapter describes adaptations of the external behavior of a component independently of a specific API. When a composition adapter is applied on a composition of components, we are able to verify whether this makes sense. Moreover, we are able to automatically insert the adaptations described by the composition adapter into the composition pattern. These algorithms are based on finite automata theory.

The next section describes the context in which this research is conducted, namely our current component based approach. The documentation of components and composition patterns is explained in more detail. Section 3 introduces the composition adapter and shortly sketches the algorithms necessary to automatically insert a composition adapter into a composition pattern. Section 4 presents the tool support that implements these ideas. After a short discussion of related work, we state our conclusions and describe our future work.

2. RESEARCH CONTEXT

We mainly focus our component based research on lifting the abstraction level for component based development. We want to realize the plug and play idea of component based development. Therefore, we propose to document components with usage scenarios that specify how to use the component. A usage scenario is expressed by a special Message Sequence Chart (MSC) [4]. The main difference with a normal MSC is that the signals are taken from a limited set of pre-defined semantic primitives. Each of these signals is also mapped on the concrete API that performs them. So the documentation of a component is both abstract and concrete. Figure 1 illustrates a usage scenario of a generic TCP/IP network component. One participant of the usage scenario represents the component and the others represent the environment participants the component expects. In this case, there's only one environment participant, namely the NetworkUser participant. This usage scenario documents that the

network component either expects data to send over the network or submits events to the NetworkUser environment participant. The network component submits an event when it received data, when the connection is established or when it is disconnected.

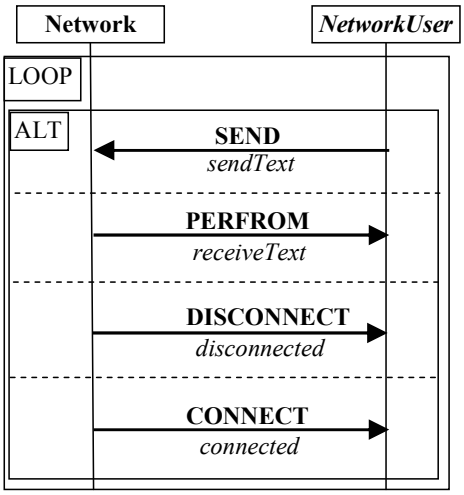

Figure 1: Usage scenario of Network component.

In addition, we introduce explicit and reusable composition patterns. A composition pattern is an abstract specification of the interaction between a number of roles and is also expressed by an MSC. The signals between the roles come from the same limited set of semantic primitives. This allows us to compare the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic game composition pattern. This composition pattern specifies the interaction between three roles: the Network, GameGui and Checker roles. One of the applications of this game composition pattern is a distributed scrabble game. The checker role is than filled by a dictionary component that is used to verify the validity of a word. The GameGUI role is filled by a dedicated Scrabble user interface component. The network role can be filled by the network component of Figure 1.

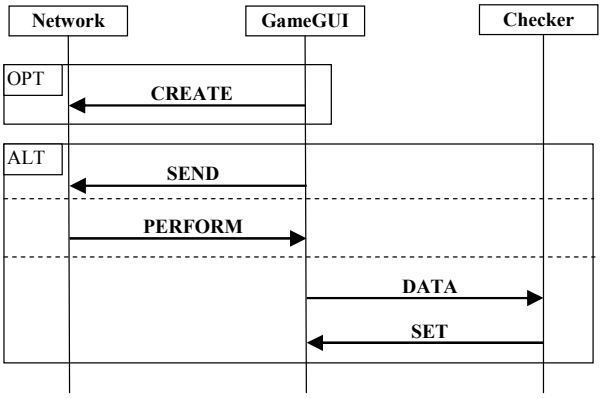

Figure 2: Generic game composition pattern.

The documentation of components and composition patterns allows us to automatically check compatibility of a component with a role. Glue-code that constraints the behavior of the components and that translates syntactical compatibilities is also generated automatically. These algorithms are based on finite automata theory. In this paper we do not go into the details of these algorithms. The interested reader is referred to [5,6,7].

3. COMPOSITION ADAPTERS

3.1 Introduction

Some aspects cannot be cleanly modularized using our current component based approach. Typical examples of such aspects are logging or synchronization. We encountered a more complicated aspect in the SEESCOA[1] research project. In this project we want to verify the quality of service of component based applications. More specifically, we would like to check both statically and dynamically whether a component based application satisfies certain timing constraints. Run-time checking of timing constraints turns out to be a crosscutting concern. If we want to check timing constraints dynamically using our current concepts, we have to alter every composition pattern individually in the same way. Of course, when the application goes into the production phase, we do not want to keep the dynamic timing aspect into the application. Consequently, we have to alter all the involved composition patterns again to remove the timing aspect. To solve this problem, we introduce the concept of a composition adapter.

3.2 Documentation

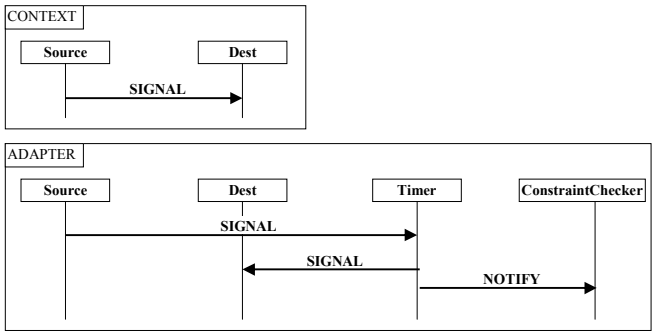

Figure 3: Dynamic timing checker composition adapter.

A composition adapter is able to describe adaptations of the external behavior of a component independently of a specific API. A composition adapter is also documented by a special kind of MSC and consists of two parts: a context part and an adapter part. The composition adapter we use to modularize the timing aspect is depicted in Figure 3. The context part describes the behavior that will be adapted. This can be as simple as one signal send as in Figure 3, but can very well be a full protocol. The adapter part describes the adaptation itself. In the case of the dynamic timing composition adapter every signal between the source and destination role will be re-routed through a Timer role. The Timer role is responsible for taking a timestamp and notifies the ConstraintChecker role. The ConstraintChecker role has a small database of timing constraints and verifies whether every signal it

[1] The SEESCOA (Software Engineering for Embedded Systems using a Component-Oriented Approach) IWT project is funded by the Flemish government. For more information see: http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/

is notified of does not violate a constraint. To minimize the disruption of the system, the component that will be mapped on the ConstraintChecker role could do the verification process offline and/or run on a different CPU.

3.3 Applying a composition adapter

When the component composer applies a composition adapter onto an existing composition pattern, the context roles of the composition adapter have to be mapped onto roles of the composition pattern. For example, suppose we want to time the communication between the GameGUI and Checker roles of the composition pattern in Figure 2. Then we would have to map the Source role of the timing composition adapter of Figure 3 onto the GameGUI role of the composition pattern. Likewise, the Dest role has to be mapped onto the Checker role. The result will be that the DATA signal is not send directly to the Checker/Dest role but is first send to the Timer role. After sending the DATA signal to the Checker/Dest role, the ConstraintChecker role is notified.

Inserting a composition adapter seems obvious from the example explained above. In this example, merely syntactically scanning the affected composition pattern would do the job. However, when the context part of the composition pattern specifies a full protocol, a more involved algorithm is needed. Therefore, we developed an algorithm in three steps based on finite automata theory. In this paper, the algorithm is only shortly sketched. A more elaborate explication of the algorithm can be found in [8]. The algorithm does not work directly on MSC's but on Deterministic Finite Automata (DFA). The transformation of an MSC to a DFA is a standard process and described in literature [9]. The first step is a verification phase. This means searching all paths in the affected composition pattern that correspond to the context part of the composition adapter. If there are no matching paths, the application of this composition adapter makes no sense. In the second step, we insert the adapter part of the composition adapter in the composition pattern at the paths that match with the context part. The last phase consists of removing all paths that match with the context part. To this end, we calculate the difference automaton between the automaton resulting from the previous phase and a special version of the context part.

4. TOOL SUPPORT

The work described in this paper has been implemented in a prototype tool called PacoSuite. PacoSuite is entirely written in JAVA and consists of two applications, PacoDoc and PacoWire. PacoDoc is a graphical editor that allows drawing, loading and saving of component documentation, composition patterns and composition adapters. The PacoWire tool is our actual component composition tool and implements the algorithms we developed in our work [5,6,7,8]. It uses a pallet of components, composition patterns and composition adapters. The tool allows dragging a component on a role of a composition pattern. The drag is refused when the component does not match with the selected role and optionally mismatch feedback is given to the user. A composition adapter can be visually applied on a composition pattern. The tool checks whether the application of the composition adapter makes sense and automatically inserts the composition adapter into the composition pattern. When all the component roles are filled, the composition is checked as a whole and glue-code is generated. Figure 4 shows some screenshots of our tool.

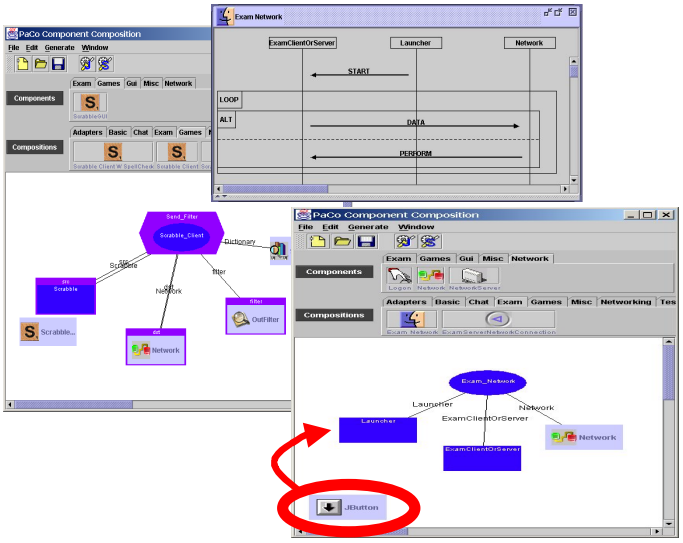

Figure 4: Screenshots of PacoSuite. At the top-right a screenshot of PacoDoc, our documentation tool is shown. At the lower-right, our actual component composition tool called PacoWire is shown. In this screenshot, the component composer is about to map a component on a role of the composition pattern. The leftmost shot shows a composition adapter that is applied on a composition pattern.

5. RELATED WORK

Although combining AOSD ideas with component based development is a rather new research direction, some approaches are already emerging. An interesting approach is event based AOSD [10]. Similar to the composition adapter approach they allow specifying an aspect on a full protocol of events.

The aspectual component approach [11] proposes a new component model to be able to specify crosscutting concerns. The aspects are weaved into the components using binary code adaptation techniques. The aspectual component approach improves on the composition adapter idea because aspects that alter the internals of a component can be specified. On the other hand, 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.

Filman [12] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. The dynamic injector approach is very similar to our composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.

6. CONCLUSIONS AND FUTURE WORK

Using composition adapters, we are able to cleanly modularize crosscutting concerns in our component based context. Composition adapters can be verified and inserted automatically in a composition of components. We improve on current aspect-oriented approaches as the joinpoints where the composition adapter will be applied are specified by a full protocol instead of a mere set of methods. An important feature of a composition adapter is that the adaptations are described independent of a concrete API, making them reusable. Consequently composition adapters still preserve the high abstraction of our visual component composition methodology. However, this approach is only able to alter the exterior behavior of components by re-routing or ignoring their messages. As a consequence, concerns that require adaptations of the interior behavior of a component cannot be specified.

To be able to alter the internals of a component we have to use an aspect-oriented implementation language. There already exists a wealth of generic approaches to separate crosscutting concerns in an object-oriented context. Well known approaches include AspectJ [13], composition filters [14] and HyperJ [15]. However, most of these approaches are not very well suited to be used in a component based context for several reasons. First, components interact in a well-defined manner (e.g. JAVA Beans interact by posting events to interested listeners), so aspects should be able to declare joinpoints specific for the component model. For the JAVA beans component model, this means that it should be possible to declare joinpoints on events. Secondly, components come from different vendors and are not explicitly created to work with each other. In order to make the aspects reusable, the declaration of the aspect behavior has to be separated from the concrete interface of the base component. This means that it should be possible to declare abstract joinpoints in the aspect specification. At aspect weaving time, the abstract joinpints are connected to concrete joinpoints in the components. Finally, source code from third-party components is often not available, therefore source code weaving becomes unfeasible. In addition, source code weaving is also unsuited for enabling the dynamic weaving and unweaving of aspects.

To solve the problems described above, we envision a new aspect-oriented implementation language tailored for the component based field. The language will be able to specify joinpoints specific for the component model. Explicit and reusable connectors connect the abstract joinpoints in the aspect declaration to concrete joinpoints in the components. The aspects are weaved into the components using binary code adaptation techniques. We already conducted experiments in component adaptation for JAVA by directly acting on the byte code instead of the source code. This has resulted in a first prototype aspect-oriented implementation language. In a next phase, we plan to use this aspect-oriented programming language as an implementation for a composition adapter. In this way, we are able to specify concerns that alter the internals of a component at a component based design level. This enables a seamless integration with our current component based methodology.

7. ACKNOWLEDGMENTS

We owe our gratitude to Dr. Bart Wydaeghe who developed the component based methodology during his PhD research. We also want to thank him for his interesting feedback and participation in this research. In addition, we like to thank Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Since october 2000 the author is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in flemish: “Fonds voor Wetenschappelijk Onderzoek”).

8. REFERENCES

[1] Szyperski, C. (1997). *Component Software; beyond Object-Oriented Programming.* Addison-Wesley.

[2] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[3] 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.

[4] ITU-TS. ITU-TS Recommendation Z.120: *Message Sequence Chart (MSC).* ITU-TS, Geneva, September 1993.

[5] Wydaeghe, B. *PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios.* PhD Thesis, available at: http://ssel.vub.ac.be/Members/BartWydaeghe/Phd/member_phd.htm

[6] Vanderperren, W. and Wydaeghe, B. *Towards a New Component Composition Process.* In Proceedings of ECBS 2001, April 2001.

[7] Wydaeghe, B. and Vandeperren, W. *Visual Component Composition Using Composition Patterns.* In Proceedings of Tools 2001, July 2001.

[8] Vanderperren, W. and Wydaeghe, B. *Separating concerns in a high-level component-based context.* EasyComp Workshop at ETAPS 2002, April 2002. To be published.

[9] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Second ed. 2001.

[10] R. Douence, O. Motelet, M. Südholt *A formal definition of crosscuts.* Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS.

[11] 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.

[12] Filman, R.E. *Applying Aspect-Oriented Programming to Intelligent Synthesis.* Workshop on Aspects and Dimensions of Concerns, 14th European Conference on Object-Oriented Programming, Cannes, France, June 2000.

[13] Kiczales G. et al. *An overview of AspectJ.* In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18--22 June 2001.

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

[15] H. Ossher and P. Tarr. *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.