EuroPLoP 2003 Focus Group: Patterns for Component Composition and Adaptation

Uwe Zdun
Department of Information Systems, Vienna University of Economics, Austria
zdun@acm.org

Markus Voelter
voelter - Ingenieurbüro für Softwaretechnologie, Germany
voelter@acm.org

Introduction and Goal of the Focus Group

Components should provide reusable, black-box building blocks as the primary commonality aspect. Many component approaches are proposed, implementing this goal in different ways. In these different component approaches there is also a strong focus on component composition and adaptation – to support variability of component architectures. Again a variety of different techniques is used for implementation. But in contrast to black-box component abstractions, these variation aspects are not well defined for practical purposes yet.

The goal of the focus group was to provide a basis for better understanding component composition and adaptation techniques through patterns. We wanted to integrate existing component composition and adaptation patterns, identify new patterns or pattern mining fields, and eventually develop a pattern language with clear sequences and alternatives for component composition and adaptation. For this first focus group on the topic we concentrated on pattern classification and integration into categories.

Problem Overview

There are many definitions of the term component. In the focus group we have considered the definition by Szyperski [Szy97] as a starting point:

> *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.*

This definition is quite broad and includes as diverse entities as server components (e.g. EJB, CCM, COM+), Java Beans, component frameworks in scripting languages (e.g. Tcl, Python, Perl, Visual Basic), Active X controls, C libraries with distinct

APIs, etc. Note that the industrial understanding of the term component is often still quite different to an "ideal" academically understanding: industrial components are often of a large scale, with no enforced boundaries, and only loosely defined interfaces. Thus the notion of a component is not really well defined for practical purposes because the term is used to denote many different things (see also [Voe03]).

Still, all named component approaches have a strong focus on component composition in common, as well as adaptation to the component context. Yet the composition and adaptation approaches of different models are quite diverse, including: component wrapping, container-managed persistence, scripting, message interception and indirection, aspect-oriented approaches for component composition, program generation and transformation, and many more.

Existing Patterns for Component Composition and Adaptation

The diversity in different component composition and adaptation approaches can well be captured by existing software patterns, even though some of them are not originally documented in the domain of component composition and adaptation. We will survey quite a few component composition and adaptation patterns in this section.

The POSA2 book [SSRB00] contains patterns for building networked object systems. Quite a few of these patterns can well be applied for component composition and adaptation as for instance:

- The Component Configurator pattern allows for runtime deployment and un-deployment of components.
- The Interceptor describes how to intercept an invocation and thus allows services to be added transparently to an application.
- The Wrapper Facade pattern can be used to wrap legacy components, such as C libraries, with an object-oriented interface.
- The Extension Interface pattern allows multiple interfaces to be exported by a component.

The patterns in the Server Component Patterns book [VSW02] in first place describe how server component architectures such as EJB are constructed. A number of patterns also describe how components are composed or adapted. Annotations allow for configuring a component declaratively. A Glue Code Layer is a generated piece of code that composes the Component Implementation and the Container. There are three patterns describing how components can be packaged for deployment: Component Installation, Component Package, and Assembly Package.

Some patterns for flexible component architectures are provided in [GZ02]. Component Wrapper describes how components can be wrapped into a system,

regardless how they are implemented. Explicit Export/Import describes how to make both interfaces of a component – the components that a component uses and the services that it provides – explicit to get traceable component architectures. Message Interceptor describes how to generally intercept messages in object-oriented systems. Message Interceptors are used to extend and adapt components.

The Component Interaction Patterns in [Esk99] describe general component interaction rules. To reduce component dependencies, an Abstract Interaction protocol between components can be defined. A Component Bus lets components interact without direct links between them. A Component Glue is a generic Adapter or Mediator between component peers. Third Party Bindings remove links established during the implementation of components. Consumer/Producer introduces a Component Wrapper [GZ02] between a using component and a number of service provider components.

The Patterns for Scripted Applications [Pry03] contain patterns that are relevant for Scripted Components. Among them is Glue Code – the typical use of scripting code in the component composition context: a script is used for composing components rapidly. The pattern Configuration Script describes a second typical use of scripting in the component composition and adaptation domain: the script is used for configuring the component in a behavioral fashion.

The patterns for structure and dependency tracing, described in [Zdu03], and the patterns for aspect-oriented composition frameworks in [Zdu04] are often used for component composition and adaptation. Metadata Tags is used for specifying configuration options. Command Language is used for behavioral configuration – in this context it can be seen as a generalization of the pattern Glue Code [Pry03]. Byte-Code Manipulator is used for load-time adaptation of components. Parse Tree Interpreter is used for compile-adaptation, and Indirection Layer/Message Interceptor are used for runtime adaptation. All three techniques are used for introducing aspect-oriented solutions into component frameworks.

Component Composition and Adaptation Techniques

We have identified the following important techniques in the component composition and adaptation domain as categories for component composition and adaptation patterns. We directly group the patterns described above into these categories:

- *Component Deployment:* Component Configurator, Component Installation, Component Package, Assembly Package.
- *Component Adaptation:* Interceptor, Message Interceptor, Byte-Code Manipulator, Parse Tree Interpreter, Indirection Layer.

- *Interfaces and Interface Descriptions with other Components:* Explicit Export/Import, Extension Interface, Abstract Interaction, Component Wrapper, Wrapper Facade, Consumer/Producer.
- *Component Configuration:* Annotations, Metadata Tags, Configuration Script, Command Language, Metadata Tags.
- *Component Gluing/Scripting:* Glue Code Layer, Component Glue, Component Bus, Third Party Binding.

Open Issues

Even though the patterns explain well how aspect-oriented solutions (AOP) can be applied in component frameworks (see [Zdu04]), the patterns do not describe how to develop successful AOP solutions for component architectures.

Some component aspects that go beyond the standard component properties are not yet covered in the interface description patterns. Examples are: QoS Tracking, QoS Specification (required, provided), Scheduler/Message Queue, and Orchestration. Orchestration is in part captured by the category component gluing/scripting, but not the aspect "how to orchestrate" components.

The use of generators in component frameworks is only in parts captured by patterns yet (only Glue Code Layer [VSW02] deals with this issue). To use of generators for reuse – as in model-driven architectures (MDA) for instance – instead of component abstractions is also not documented in pattern form yet. Also patterns for templates for components and component interactions used in such architectures are missing.

Focus Group Participants

The following participants have been at the focus group: Louis-Pierre Gagnaux, Stefan Hanenberg, Wolfgang Herzner, Chisanga Mwelwa, Andy Longshaw, Michael J. Pont, Kristian Elof Sørensen, Markus Voelter, Charles Weir, and Uwe Zdun.

References

[Esk99] P. Eskelin: Component Interaction Patterns, Proceedings of 6th Conference on the Pattern Languages of Programming (PloP), Illinois, USA, 1999.

[GZ02] M. Goedicke and U. Zdun: Piecemeal Legacy Migrating with an Architectural Pattern Language: A Case Study, In Journal of Software Maintenance and Evolution: Research and Practice, 14:1-30, 2002.

[Pry03] N. Pryce. Patterns for Scripted Applications.

http://www.doc.ic.ac.uk/~np2/patterns/scripting/index.html, last visited: 2003.

[Szy97] C. Szyperski. Component Software – Beyond Object-Oriented Programming. ACM Press Books. Addison-Wesley, 1997.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J. Wiley andSons Ltd., 2000.

[Voe03] M. Voelter. A taxonomy for components. JOT, 2003, to appear. http://www.voelter.de/data/articles/ComponentTaxonomy.pdf

[VSW02] M. Voelter, A. Schmid, E. Wolff: Server Component Patterns, Wiley, 2002.

[Zdu03] U. Zdun: Patterns of tracing software structures and dependencies. In Proceedings of EuroPlop 2003, Irsee, Germany, June 2003.

[Zdu04] U. Zdun. A pattern language for the design of aspect languages and aspect composition frameworks. Draft; accepted for publication in IEE Proceedings Software, special issue on Unanticipated Software Evolution, 2004.