Concepts For Developing Component-based Systems

R. Schmidt, U. Assmann
Forschungszentrum Informatik (FZI) and Universität Karlsruhe
email: rschmidt@fzi.de, assmann@ipd.info.uni-karlsruhe.de

The problem

Increasing the productivity of software system development and augmenting the flexibility of software systems to react to business process changes has become a dominant concern for companies competing in the global marketplace. Taking their clues from traditional production techniques, IT systems should be constructed from prefabricated, easily marketable components that can be widely reused. An example of such a product is the ubiquitous screw, which is used in a large number of contexts. This is possible, because the screw has a certain amount of context-independence. Context independence means that a component is easily transferable from its development context (the screw factory) to a wide variety of application contexts (cars, ships, planes) and it can be easily replaced by other components with similar functionalities but different or better qualities (by a screw from stainless steel). Following, this example, software-units should be context-independent in order to build and evolve component-oriented systems [CiSc96], [Schm97]. And indeed, modern software architectures based on ActiveX/DCOM [Chap96], CORBA [OMG] or Java Beans [Beans] do support the development of systems from independently developed software units (components). However, there is no widely-accepted method for identifying the components and how to compose them to applications.

Components- and component-based frameworks

The concepts "component" and "component-based framework" which can be found in technologies like Enterprise Java Beans [Beans], ActiveX/DCOM [Chap96] should not be confused with the components and connectors found in architecture definition languages [Medv97], which are mentioned later. Component-based frameworks such as ActiveX/DCOM aim at enabling the cooperation of independently developed software-units, called components, across computer and network boundaries. Architecture definition languages [Med97] and architectural styles are description models and aim at the structural description of software systems. They try to give a more abstract view of software systems than object-oriented methods do, for example, [GaSh93] and allow analyses with broader view.

Component-based frameworks not only support the cooperation of components across computer and network boundaries, they also enable the independent evolution of the components: Components can be easily replaced by other components, offering new or enhanced functionality. Furthermore, component-based frameworks provide services for the integration of databases, file systems etc. JavaBeans Enterprise [Beans], for example, allows the remote interoperation of beans over Remote Method Invocation (RMI). The Java Database Connection (JDBC) provides the integration of relational databases. Other services which can be used from JavaBeans Enterprise are – for example - the Java Transaction Service (JTS), the Java Naming and Directory Interface (JNDI), the Java Message Services (JMAPI) etc.

Two concepts support the evolution in component-based frameworks: The first concept is that of strong interfaces in the components. They completely hide the implementation and the implementation model of the component (In ActiveX/DCOM for example, interfaces are defined at the binary level, although more comfortable language adaptations exist). Therefore, a component can be replaced by another one, as long as the same interface is supported. The second concept is implicit interface invocation. In a component-based framework, the component supporting an interface is never directly

addressed. Directly addressing the component would imply that the user of the component has to know the identity and the location of a component. Therefore, the user of a component only specifies the interface required and the component-based framework returns a reference to a component providing the interface. This reference may be a component but also an representative, such as a proxy. The user of the component, however, does not see any difference. The information about the interfaces and the location of a component implements is stored in a registry or repository and not in the components themselves. An application built from components differs from applications built with conventional programming. The components do not address one another directly, but address each other using the indirection mechanisms described above. Therefore, it is possible to exchange components flexibly and even to relocate them to another computer using implicit invocation. The components composing the application, may not form a single executable, but can be distributed across different computers.

Because the implementation details of a component are inaccessible for the user, there have to be mechanisms to provide information about the component's specification. The same is necessary for the adaptation of a component to individual requirements which must be possible without knowing about its implementation. To achieve this goal, components offer two mechanisms, *introspection* and *specialization*. *Introspection* allows one to gain knowledge about the component's interfaces without knowing its implementation. For example, Java Beans provides two mechanisms for introspection, reflection and the BeanInfo class. *Reflection* allows to know about all methods of a component without further programming. The BeanInfo class gives complex information about a Bean component, but has to be implemented explicitly. *Specialization* allows a component to be changed without access to its implementation. Specialization in Java Beans is supported by property sheets and customizers. Property sheets offer a simple specialization mechanism which sets the parameters of the Bean. Complex specializations can be done by customizers. Both introspection and specialization are used by the component weaver to combine the aspect implementations. The introspection mechanisms give the component weaver information about the connection points of the component, that means, how other component implementations can be connected to the component. The specialization mechanisms are used to connect one component to another component.

Designing component-based systems

At first glance, one would expect object-oriented architectures to meet the needs of component-oriented architectures. After all, the properties of components and those of objects appear to be quite similar. If this were indeed so, one could employ a wealth of established methods and techniques for component-based architectures. However there are some methodical weaknesses of object-orientation, which become obvious in large systems of independent evolving components. One example is the representation of interaction protocols. They are not independent entities separate from the objects, but dispersed amongst all objects participating in the interaction [LoWa95]. Therefore changes to the interaction protocols often require changes to several objects and hamper system evolution. Keeping such changes consistent is a further problem [AWBB93]. The overall control flow can only be comprised of the combined control flows of individual objects. Furthermore, such objects with an embedded part of an interaction protocol can only be reused in a new context, if the same interaction protocol is used by the other objects. An example of the inadequacies of object-oriented methods is found in the implementation of business processes. Object-oriented methods intermix single operations with the global control flow responsible for the sequence of the operations. Neither is it easy to reuse objects in other business processes, because they also contain a part of the global control flow; nor is it easy to change the global control flow, because it is embedded in a multitude of objects.

In response, several attempts have been made to fix these deficiencies. Alliances [LoWa95] separate interaction protocols from the objects. Composition filters [AWBB93] realize interaction protocols by filtering the messages sent between the objects and by providing support for error detection and synchronization. Architecture definition languages [Med97] describe software systems as combination of components, which provide the basic functionality, and connectors, which describe the relationships of the components, architecture definition languages attempt to give a conceptual description of the system. By this means, it is possible to separate a application context-independent core functionality

represented by components, from context dependent functionality represented by connectors. For example, connectors can be used to separately model interactions, such as events, pipes etc [GaSh93].

The aforementioned weaknesses of object-orientation are symptomatic of a much broader problem, the "tangling of aspects" as described in [KILL97], [Kicz96]. Embedding the interaction protocols into objects can be seen as the mixing of the interaction aspect with other aspects, such as the operational one. Changes to one aspect causes also many side-effects to the implementation of other aspects. Therefore, the concept of Aspect-oriented Programming [KILL97] proposes to separately specify and implement the different aspects of an application and combine the implementation through an so-called aspect weaver. Aspect-oriented Programming goes beyond untangling the interaction protocols from the objects and extends the idea to other aspects, such as error handling, distribution, etc., which usually are also intermixed when using traditional object-oriented methods.

Position

Our thesis is that application development for component-oriented systems should start with an aspect-separated model. Ideally components should implement only functionality belonging to one aspect of an application, and applications should be composed of aspect-separated components. By applying the concept of aspect-oriented programming there is a much better chance that components become truly reusable, because they have to fulfill only the requirements of one aspect and not a combination of several ones. Component-oriented systems built in an aspect-separated manner can then be expected to become more flexible and evolution-transparent, because changes which concern only one aspect, only influence implementations of one aspect. To combine the components, the aspect weaver [KILL97] should be adapted for combining components in form of a component-weaver.

We argue that the decomposition of the problem domain into aspects, as proposed by [Berg97] will play an important role in applying the idea of aspect-oriented programming to component-oriented systems. Aspect-oriented programming should be combined with aspect-separated domain models, such as workflow-models [Jabl95]. Applying the separation of aspects found in workflow-management-systems will provide clues as to how constitute components which could then be utilized in a multitude of business processes. On the other hand, workflow-management systems may profit from component-oriented systems. Our goal is to fuse component-oriented systems and aspect-oriented programming paying special consideration of domain-specific aspect-oriented models such as workflow models, in order to obtain the best of both worlds.

References

[AsSc97] U. Aßmann, R. Schmidt: Towards a Model For Composed Extensible Components. Workshop Foundations of Component-Based Systems, Proceedings, Zurich, Switzerland September 26, 1997

[AWBB93] M. Aksit. , K. Wakita, J. Bosch , L. Bergmans and A. Yonezawa : Abstracting Object Interactions Using Composition Filters. In Object-Based Distributed Programming, R. Guerraoui, O. Nierstrasz and M. Riveill (eds.), LNCS 791, Springer Verlag 1993

[Berg97] L. Bergmans: Aspects of AOP: Scalability and application to domain modelling. TRESE project, University of Twente & STEX. http://www.parc.xerox.com/spl/projects/aop/aop-meeting-pps/bergmans.html

[Beans] Javasoft: Java Beans Specification 1.0 A. http://splash.javasoft.com/beans/-beans.100A.pdf

[Chap96] D. Chappell: Understanding ActiveX and OLE. Microsoft Press. Redmond 1996

[CiSc96] O. Ciupke, R. Schmidt: Components As Context-Independent Units of Software. WCOP 96, Linz 1996. Special Issues in Object-Oriented Programming. Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP96. Dpunkt.verlag, Verlag 1996

[Jabl95] Jablonski, S.: Workflow-Management-Systeme. International Thomson Computer Press. Bonn 1995

[Kicz96] G. Kiczales: Aspect-oriented programming. ACM Computing Surveys, 28(4), Dec. 1996.

[KILL97] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C. V. Lopes, C. Maeda, a. Mendhekar: Aspect-Oriented Programming. Position Paper from the Xerox Parc Aspect-Oriented Programming Project.

[LoWa95] P.C. Lockemann, H. D. Walter: Object-Oriented Protocol Hierarchies for Distributed Workflow Systems. In [PaTo95].

[OMG] http://www.omg.org

[PaTo95] R. Pareschi, M. Tokoro: TAPOS Theory And Practice of Object Systems. John Wiley, New York. Volume 1(1) SPECIAL ISSUE: 1995 European Conference of Object Oriented Programming

[ScAs98] R. Schmidt, U. Assmann: Compflow. ACM Symposium on Coordination languages. Atlanta 28.2.98.

[Schm97] R. Schmidt: Component-based systems, composite applications and workflow-management. Workshop Foundations of Component-Based Systems, Proceedings, Zurich, Switzerland
September 26, 1997

[WfMC] Workflow Management Coalition: http://www.aiai.ed.ac.uk/project/wfmc/