Position Paper

Componentware – The Big Picture*

Klaus Bergner, Andreas Rausch, Marc Sihling

Institut für Informatik
Technische Universität München
D-80290 München
http://www4.informatik.tu-muenchen.de

March 10th,1998

1. Introduction

The goal of componentware is to carry the old dream of building software systems by assembling pre-fabricated components to undreamt brilliancy. Although there is a variety of technical concepts and tools for component-oriented software engineering, the successful model from the building industry could not be transferred fully to software development yet. In our opinion, this is mostly due to the lack of a suitable methodology. Such a methodology should consist of the following parts:

- A well-understood and formally founded *component concept* is needed as a basis. The concept has to be simple, but sufficiently powerful to model the essential concepts and development techniques of the existing technical component approaches.
- Based on the component concept, *description techniques* for components are needed. They are necessary for communication with the customer and between the developers. Examples for such description techniques are the notations provided in the Unified Modeling Language (UML) [UML97], as well as programming languages like C++ or Java. Well-defined consistency criteria allow to check the correctness of different views of a system with the help of specialized tools.
- Development must be organized according to a *process* tailored to componentware. Such a process has to be organized and structured differently compared to the creation of systems from scratch. This includes especially the assignment of tasks to different roles, for example, to a software architect who is responsible for the overall design of a system, or to component developers who develop and sell reusable components.
- The description techniques and the overlying componentware process must be supported by *tools*. These tools should at least be able to generate the system itself as well as any kind of documentation for the system. Furthermore, tools should be able to verify and prove critical properties of the system. To perform these tasks, tools must combine and implement the three former points.

In the following sections we cover these aspects in more detail.

2. Componentware – Characteristics and Concepts

Componentware is the art of building systems from components. Hence, the most important concepts in componentware are *components* and their *interfaces*.

The interfaces of a component is particularly important for the composition and customization of components by users: They allow them to find suitable components and to understand their purpose, functionality, usage, and restrictions. The description of a component interface consists of

- a *signature part*, describing the operations provided by a component, and, based on that
- a *behavior part*, describing the component's behavior.

However, most description techniques for interfaces, such as the Interface Definition Language (IDL), are only concerned about the signature part.

Components can *implement* (resp. *export*) one or more interfaces, and they can also *use* (resp. *import*) interfaces from other components. Hence, export interfaces correspond to the services a component provides to the environment, and an import interface corresponds to the services a component needs from the environment to implement the exported services. Together, import and export interfaces of all components constrain the possible connections between components. To build a system of components, one needs to define a mapping by connecting compatible import and export interfaces.

We can distinguish the concept of an instance from the concept of a type, analogous to object-orientation. Component types are not necessary in some approaches, for example, with large legacy components, where only one component instance of a type exists. However, the type concept is useful to get a clear and uniform framework of notions suitable also for approaches with explicit types like Java Beans [SUN97].

The connection structure on the type level is usually static and does not change during the runtime of a system, but only during development. The connection structure of the instances may not violate the constraints given on the type level, but is more dynamic, as component and connection instances can be created or destroyed at

* This paper is an executive summary of the paper „Componentware – The Big Picture" written by K. Bergner, A. Rausch, M. Sihling. Both papers are a work of the research project „A1 Methods for Component-Based Software Engineering", a part of „Bayrischer Forschungsverbund Sofware-Engineering (FORSOFT)".

runtime under control of the system itself. Of course, certain variations are possible here with respect to the admitted connection structures and their possible changes (for an example see [BKR+98]). A further level of extension can be introduced by permitting dynamically changing interfaces, similar to the DCOM interface concept [Cha96].

3. Description Techniques for Components

So far, we have focused on providing a model for components. For describing them, we mainly use and adapt existing, well-known description techniques from UML [UML97], Fusion [CAB+94], and Catalysis [DW95].

Describing interfaces of components means to describe the *black-box view* of components. To specify the signature part of interfaces *component interface diagrams* (CIDs) can be used (cf. [HRR98]). CIDs are an adapted version of UML class diagrams, where each box represents an import or export interface of the enclosing component. Additionally, with enhanced CIDs one can specify multiplicity of interfaces and navigation between several interfaces supported by one component. CIDs can be easily mapped into existing componentware infrastructures, like CORBA [OH97], ActiveX [Cha96] or Java Beans [SUN97] (cf. [HRR98]). Additionally, CASE tools like Rational Rose could generate CORBA or DCOM IDL-Files from CIDs. The behavior part on the interface level can be described, for example, with state machines or sequence diagrams.

To describe the internal implementation of a component, the so-called *glass-box view*, one can, for example, use class diagrams, state machines, and sequence diagrams. The glass-box view should also describe the dynamic change of the instance and connection structure of a component's implementation. In general, the existing description techniques are not sufficient in that area, and new techniques must be found.

4. Componentware Methodology

On top of the description techniques a component-based development method is needed. This method should take care of the following points:

- *Adaption of proven methodology*: Componentware does not introduce a totally new paradigm, but rather builds on existing, well-proven techniques. Hence, the methodology for componentware should not be totally new, but adapt, improve, and combine well-known object-oriented methodologies.
- *Combining bottom-up and top-down development*: As practical experience has shown, neither a pure top-down nor a pure bottom-up approach is adequate for the development of large applications. To overcome the deficiencies of both extremes, the componentware approach has to combine top-down and bottom-up development as equivalent parts complementing each other.
- *Guidelines for using description techniques*: The process model should provide developers with guidelines when and how to use the different description techniques and when and how to check the resulting documents for consistency.
- *Support for reuse and customization*: Current methods do not sufficiently take into account issues of reuse and customization. Therefore, the reuse rate in practice is very low. To reach a higher rate of productivity, we have to introduce new tasks performed by persons in new roles.

Figure 1 shows our proposal for the component-based development method. The main tasks are similar to those of the waterfall model: *analysis*, *architecture design*, *component design*, and *implementation*. In contrast, we do not require that these tasks are performed sequentially, but allow to switch between them or even to perform them concurrently. However, we require well-defined interfaces between them, for example, documents, component and interface specifications or implementations, and prototypes. This is shown by the horizontal, three-dimensional two-way arrows.

Each main task consists of several subtasks. Although these phases eventually run in parallel and influence each other, there may not necessarily be well-defined interface documents. Figure 1 illustrates this fact by connected boxes, like the boxes *requirements elicitation*, *business component evaluation*, and *business component search*.

Figure 1 also includes some special subtasks (resp. task switches) where one should interact with *component vendors*. For instance, in the subtask *component search*, the *component assembler* is searching for suitable components in several component repositories provided by vendors. For some of these components, the *component assembler* will evaluate whether they match the requirements and fit into the existing architecture. Whenever components are delivered by vendors, these components have to be tested against the specification and the test scenarios worked out.

As can be seen, our focus is not on processes, but on tasks, their deliverables, and their depencies between them. This allows to use our methodology with several process models, like rapid prototyping or the spiral model, and also to walk through phases iteratively. Furthermore, developers are not forced to follow a strict bottom-up or top-down approach. Instead, a combination of top-down and bottom-up development with cyclic and iterative aspects is possible.

We also introduced new roles, illustrated by the flags in Figure 1. In our eyes, one of the central issues with componentware is especially the separation of the roles of *component vendors* and *component users*. It is a necessary prerequisite for the rise of a market of specialized, reusable high-quality components needed to build the large and highly complex systems of the future. To get more detailed information about the development method see also the full version of the paper [BKR+98].

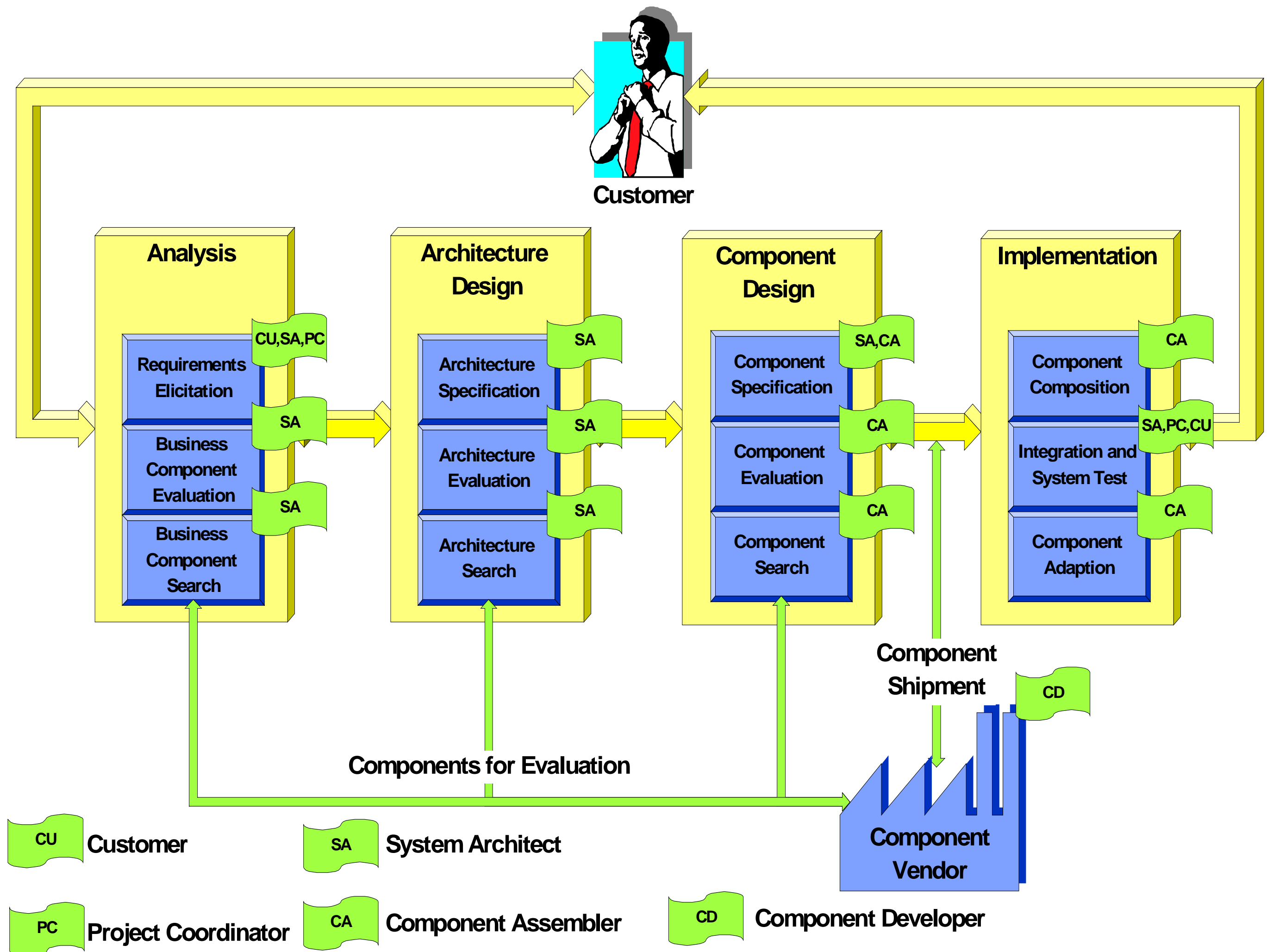

Figure 1: A Component-Based Development Method

5. Requirements for Tools Supporting Componentware

To reach the real benefits of componentware, we have to integrate the proposed concepts, description techniques, and processes into tools. Tools can be distinguished as follows:

- *Component specification*: Specification tools provide means for describing the signature and behavior part of component interfaces. To reach that goal, some of the description techniques introduced in Section 3 should be supported. Specification tools are, for example, used by component vendors to describe their product in an understandable fashion as well as by component users to specify the component they intend to buy.
- *Component composition*: Tools for component composition are already on the market, for example, SUN's Visual Java or IBM's Visual Age. However, they just implement a subset of the needs. For instance, they cannot deal with the dynamic creation and deletion of components or connections properly and do not check the validity of composition on the semantic level.
- *Component brokers*: One of the most important tasks in our method (see Section 4) is the search for already existing components. This involves specialized tools for understanding the requirements and for browsing through component catalogues or one's own legacy components.
- *Component workflow*: Tools supporting the development process can be helpful only if they are also aware of the separation of roles. Basically, this comes down to specialized workflow management.

6. Further Work

The proposed framework for componentware, the component concepts, description techniques, and the component-based development process model were defined as a result of several case studies. In the future, we will further work out and improve this framework and integrate it into tools.

7. References

[UML97] UML Group: Unified Modeling Language 1.1. Rational, 1997.
[Cha96] D. Chappell: Understanding ActiveX and OLE. Microsoft Press, 1996.
[SUN97] Sun Microsystems: Java Beans 1.01. Sun Microsystems, 1997.
[OH97] R. Orfali, D. Harkey: Client/Server Programming with JAVA and CORBA. John Wiley and Sons, 1997.
[BKR+98] K. Bergner, A. Rausch, M. Sihling: Componentware – The Big Picture. TU München, 1998.
[HRR98] F. Huber, A. Rausch, B. Rumpe: Component Interface Diagrams: Putting Components to Work. TU München, 1998.
[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: Object-Oriented Development – The Fusion Method. Prentice-Hall, 1994.
[DW95] D. D´Souza, A. Wills: Catalysis – Practical Rigor and Refinement. Technical Report, 1995.