Componentware - Methodology and Process

Klaus Bergner, Andreas Rausch,Marc Sihling, Alexander Vilbig
{bergner|rausch|sihling|vilbig}@in.tum.de

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

19th March 1999

Introduction: Componentware Methodology

Componentware is concerned with the development of software systems by using components as the essential building blocks. It is not a revolutionary approach but incorporates successful concepts from established paradigms like object-orientation while trying to overcome some of their deficiencies. Proper encapsulation of common functionality, for example, and intuitive graphical description techniques like class diagrams are keys to the widespread success of an object-oriented software development process. However, the increasing size and complexity of modern software systems leads to huge and complicated conglomerations of classes and objects that are hard to manage and understand. Those systems obviously require a more advanced means of structuring, describing and developing them. Componentware is a possible approach to solve these problems.

An analogy to the building industry illustrates a successful application of such a component-oriented approach: First, the building owner provides the architect with the functional and non-functional requirements in a more or less informal way. Examples are the number and function of rooms and the money he wants to spend. The architect then constructs a first, overall ground plan and several side views or even a computer-generated virtual model of the building. If these proposals meet the owner’s expectations, the architect will elaborate a more detailed and technical construction plan. It describes the different components of the building, like walls and windows, and how they fit together. Now the architect invites tenders for these components and evaluates their offers. At last, the ‘‘best’’ component producers get the job, place the components to the architect’s disposal, and integrate them into the building. During the whole process, the architect’s construction plan is the basis of communication between all parties working on the building.

Although there already exist a variety of technical concepts and tools for component-oriented software engineering, the successful model from the building industry was not completely transferred to software development yet. In our opinion, this is partly due to the lack of a suitable componentware methodology. Such a methodology should at least incorporate the following parts:

- A well-defined conceptual framework of componentware is required as a reliable foundation. It consists of a mathematical *formal system model* which is used to unambiguously express the basic definitions and concepts. The contained definitions and concepts should be as simple as possible, yet sufficiently powerful to capture the essential concepts and development techniques of existing technical component approaches.
- Based on the formal system model, *description techniques* for components are required. They correspond to the building plans of architecture and are necessary for communication with the customer and between the developers. Examples for description techniques are graphical notations like class diagrams and state transition graphs from modeling languages like UML as well as textual notations like interface specifications expressed in CORBA IDL, C++, or Java. Well-defined consistency criteria between the different description techniques allow to verify the correctness of different views onto a system with the help of specialized tools.
- Development should be organized according to a *process model* tailored to componentware. This includes in particular the assignment of discernible development tasks to individuals or groups in different roles, for example, a software architect responsible for the overall design of a system, and component developers who produce and sell reusable components.
- The description techniques and the componentware process model should be supported by *tools* . At least, these tools should be able to generate an implementation of the system as well as corresponding documentation. Furthermore, they could facilitate the verification of critical system properties, based on the formal system model.

A more extensive discussion about these fundamental parts of a componentware methodology can be found in [BRSV98b]. In the following sections we focus on the process model for componentware in detail. Such a process model supports system development by clearly defining individual development tasks, roles and results as well as the relationships between them. We first cover the essential aspects that distinguish a component-oriented process model from more traditional approaches. Subsequently, we introduce new development roles associated with componentware and propose a suitable process model for component users and developers. Then we discuss some specific component developer issues. A short conclusion ends the paper, referring to the strawman outline of CBSE'99 [CBS99].

Requirements for a Component-Oriented Process Model

The characteristics of a componentware methodology as described in Section 1 require a suitable process model. Such a process model should itself follow the componentware pardigm: it should consist of a box of building blocks which can be individually tailored to the specific needs of the actual project. There should also be a strong focus on reuse and architectural issues.

New Tasks and Roles

To leverage the technical advantages of componentware and to support the reuse of existing components, the introduction of new tasks accomplished by individuals or groups in new roles is immanent. This pertains to roles like *Component Developer* and

Component Assembler , and to tasks like searching for existing components and evaluating them before their integration into the overall system architecture. The initial elaboration and the continued development of such an architecture requires further tasks like architecture design to be performed by special roles like system architects.

Adaptability and Flexibility

The rigidity of traditional, prescriptive process models is widely felt as a strong drawback, and there is common agreement about the need to adapt the process to the actual needs. A flexible process model should be more modular and adaptable to the current state of the project, much in analogy to the essential properties of components and componentware systems themselves. To provide the necessary flexibility, our approach uses so-called ‘‘Process Patterns’’ (cf. [Cop94,DW98,BRSV98a]).

Combining Top-Down and Bottom-Up Development

With componentware, the successful combination of top-down and bottom-up development is essential. On the one hand, one has to take into account the initial customer requirements, breaking them down into components in a top-down fashion until the level of detail is sufficient for implementation. On the other hand, one has to reach a high reuse rate of existing components. Hence, one starts with existing, reusable components, which are then iteratively combined and composed to higher-level components in a bottom-up fashion. Obviously, neither pure bottom-up nor pure top-down approaches are practical in most cases. New process models, like the Rational Unified Process [Iva99] or the German V-Modell [IAB98] already try to resolve these both aspects by defining an iterative and incremental process.

Evolutionary Approach

The introduction of new roles and tasks is a key aspect of a process model tailored to componentware. However, this doesn’t imply that the process model in question has to be completely new and revolutionary. After all, componentware is itself an evolutionary approach based on the technical foundations of earlier paradigms like object-orientation. Therefore a proposed process model for componentware should represent an adapted and improved version of established practice. In [ABD+99] we have outlined how the German V-Modell standard can be tailored to componentware, focusing on reuse issues and process patterns.

Roles of a Component-Oriented Process Model

The distinction between the roles of *Component Vendors* and *Component Users* is a key aspect of a component-oriented development process. It is a necessary prerequisite for the rise of a market for specialized, reusable components of high quality that are needed to build large, reliable and highly complex systems. Other, more mature industrial branches have known this separation for a long time [Hin97]. We expect the following, specialized roles to evolve in the context of component-oriented software development:

Component Developer:

Components are supplied by specialized component developers or by in-house reuse centers as a part of large enterprises. The responsibilities of a *Component Developer* are to recognize the common requirements of many customers or users and to construct reusable components accordingly. If a customer requests a particular component, the *Component Developer* offers a tender and sells the component.

Component Assembler:
Usually, complicated components have to be adapted to match their intended usage. The *Component Assembler* adapts and customizes pre-built standard components and integrates them into the system under development.

System Analyst:
As in other methodologies, a *System Analyst* elicits the requirements of the customer. Concerning componentware, he also has to be aware of the characteristics and features of existing systems and business-relevant components.

System Architect:
The *System Architect* develops a construction plan and selects adequate components as well as suitable *Component Developers* and *Component Assemblers* . During the construction of the system, the *System Architect* supervises and reviews the technical aspects and monitors the consistency and quality of the results.

Project Coordinator:
A *Project Coordinator* as an individual is usually only part of very large projects. He supervises the whole development process, especially with respect to its schedule and costs. The *Project Coordinator* is responsible to the customer for meeting the deadline and the cost limit.

Process Model for Componentware

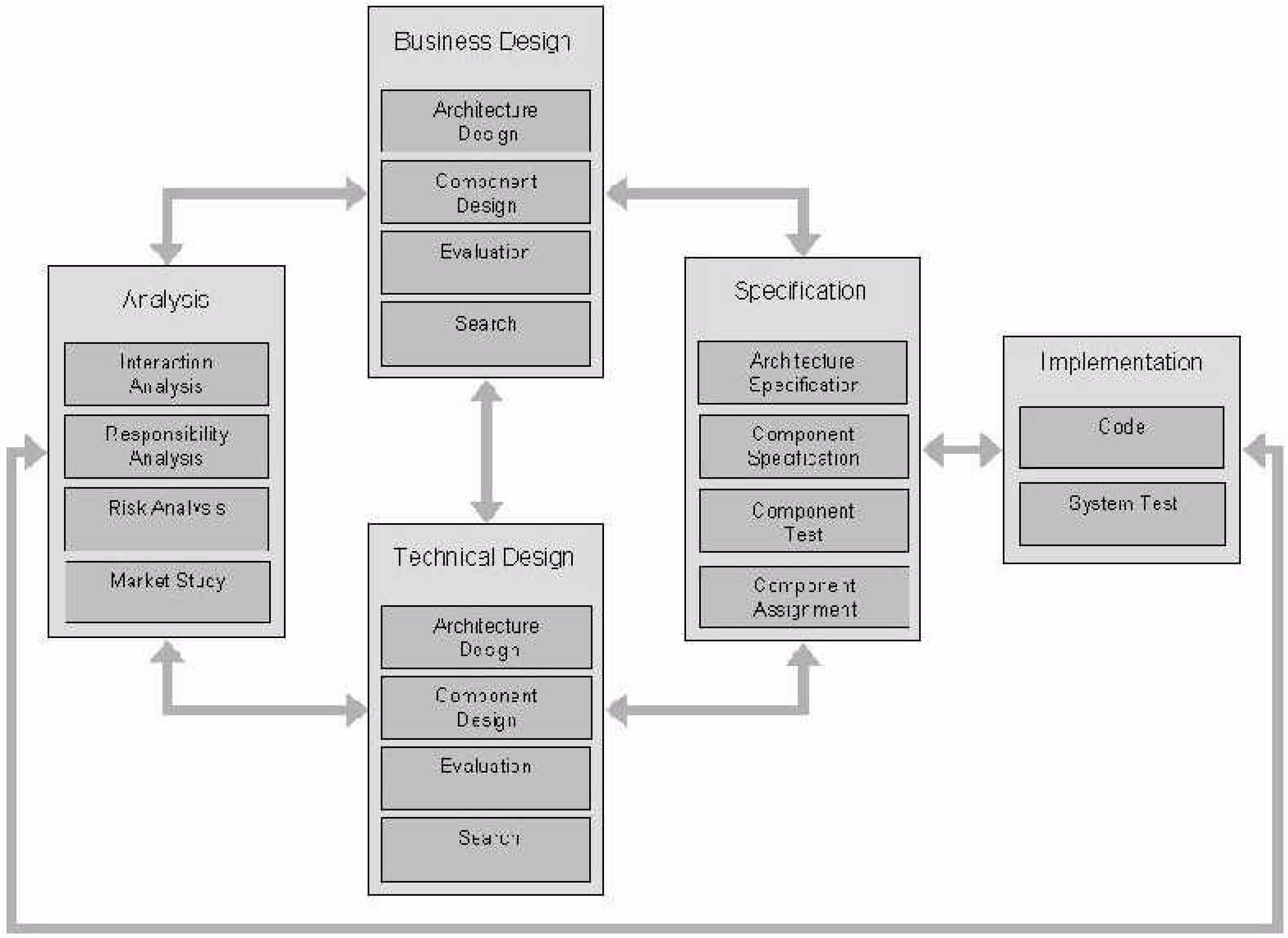
Figure 1 illustrates our proposal for a flexible, component-oriented process model. It shows the different tasks of a componentware development process. Each of these tasks produces results. Thus, a process model for componentware contains a hierarchical *task structure* resp. *result structure* . Note that the presented concepts apply to *Component Users* , i.e. the developers of component-oriented systems, as well as for *Component Vendors* shipping components to *Component Users* .

Figure 1: Component-Oriented Process Model (*figure appears on the next page*)

The main parts are resembling the phases of conventional process models although we explicitly separate business-oriented design from technical design: *Analysis*, *Business Design*, *Technical Design*, *Specification*, and *Implementation*. All main development tasks, like *Business Design*, consist of subtasks like *Architecture Design*, *Component Design*, *Evaluation*, and *Search* each of which is requiring and/or producing development results. For instance, during some task a so-called *Component Design Document* is created which may contain several diagrams using the description techniques mentioned in [BRSV98b] and which is reflected in the result structure shown in Figure 1.

The produced documents and other development artifacts serve as interfaces of the main tasks, analogous to ‘‘real’’ interfaces of software components. The connections between the tasks, namely, the consistency conditions and the flow of structured development information, are visualized by thick, grey arrows in Figure 1.

Note that there are no arrows between the subtasks (resp. subresults) in a main task (resp. result). This is due to the fact that these subtasks are even closer coupled than the main tasks


Business Design
Architecture Design
Component Design
Evaluation
Search
Analysis
Interaction Analysis
Responsibility Analysis
Risk Analysis
Market Study
Specification
Architecture Specification
Component Specification
Component Test
Component Assignment
Implementation
Code
System Test
Technical Design
Architecture Design
Component Design
Evaluation
Search

and are usually developed together. As componentware is based on reusing existing software, it is not plausible, for example, to design the technical architecture (subtask *Architecture Design* of main task *Technical Design*) without searching for and evaluating existing technical components (subtasks *Search* and *Evaluation* of the same main task).

In contrast to traditional process models, we do not define any particular order on the temporal relationship between the development tasks and their results. We believe that a truly flexible process should be adapted to the current state of the project which is partly determined by the current state of the development documents. According to this state, a given development context, and a set of external conditions, we define so-called process patterns which provide guidelines about the next possible development steps. Details about the proposed result structure and the pattern-based approach can be found in [BRSV98a]. In the following, we describe the tasks (resp. results) and involved roles in more detail.

Analysis:

The *Analysis* main result resp. task contains the specification of the customer requirements. The subresult *Interaction Analysis* is concerned with the interaction between the system and its environment. It determines the boundary of the system, the relevant actors (both human and technical systems), and their usage of the system to be developed. Contained may be parts like an overall *Use Case Specification*, a *Business Process Model*, *Interaction Specifications* including *System Test Cases*, and an explorative *GUI Prototype*.

The subresult *Responsibility Analysis* specifies the expected functionality of the system with respect to the functional and non-functional user requirements. It describes the required services and use cases of the system in a declarative way by stating *what* is expected without prescribing *how* this is accomplished. Contained are parts like *Service Specifications*, *Class Diagrams*, and a *Data Dictionary*.

The subresult *Risk Analysis* identifies and assesses the benefits and risks associated with the development of the system under consideration. In the context of componentware, this requires a *Market Study* with information about existing business-oriented solutions, systems, and components.

Note that *Analysis* usually not only covers functional and non-functional requirements, but also technical requirements restricting the technical architecture of the system to be built. While the functional requirements must be fulfilled by the *Business Design* main result, the non-functional and technical requirements must be compliant with the *Technical Design* main result. Furthermore, the *Implementation* must pass the *System Test Cases*.

Business Design:

Business Design defines the overall business-oriented architecture of the resulting system and specifies the employed business components. The subresults *Architecture Design* and *Component Design* are comparable to the *Interaction Analysis* and *Responsibility Analysis* subresults of *Analysis*. However, they do not address technical issues, but instead provide a detailed specification of the business-relevant aspects, interactions, algorithms, and responsibilities of the system and its components. *Search* corresponds to a preselection of potentially suitable business components and standard business architectures that are subject to a final selection within the *Specification* main result. Within *Evaluation*, the characteristics of the found components and architectures are balanced against the criteria identified in *Architecture Design* and *Component Design*.

Technical Design:

Technical Design comprises the specification of technical components, like database components, for example, and their overall connection architecture which together are suited to fulfill the customer's non-functional requirements. As this result deals with technical aspects of the system like persistence, distribution, and communication schemes, *Technical Design* represents a dedicated part of the development results that should be logically separated from *Business Design*.

In the context of componentware, however, the applied development principles are the same for both areas. Consequently, the involved subresults are analogous to those of *Business Design*.

Specification:

The main results *Business Design* and *Technical Design* are concerned with two fundamentally different views on the developed system. The *Specification* main result merges and refines both views, thereby resulting in complete and consistent *Architecture* and *Component Specifications*.

As said above, both *Business Design* and *Technical Design* cover an evaluation of existing components from the business and technical point of view, resulting in a preselection of potentially suitable components for the system. The *Specification* subresult *Component Test* contains the results and test logs of these components with respect to the user requirements and the chosen system architecture. Note that such tests should be performed as soon as the specification of a component is available in order to avoid problems during system integration.

Some of the desired components may simply be ordered whereas other components are not available at all and must be developed. The *Component Assignment* subresult specifies which components are to be developed in the current project and which components are ordered from external component suppliers or in-house profit centers. If a component is to be developed outside of the current project, a new, separate result structure has to be set up. Note the close correspondence between *Architecture Specification* and *Component Specification* on the one hand, and *Interaction Analysis* and *Responsibility Analysis* on the other hand. It allows for a clear hand-over of a component specification to a component developer outside the project.

Implementation:

The most important subresult of the *Implementation* main result is of course the *Code* of the system under consideration. It comprises source code as well as binary-only components. The other subresult covers the *System Test* results.

Note again that all subtasks mentioned in the above sections may be performed concurrently and influence each other mutually. For instance, it might be advisable to implement and test critical subsystems early in order to reduce the development risk.

Component Developer Issues

A *Component Developer* implements and ships components to his customers, the component users. These component users may be end-users, i.e. *Project Coordinators* , *System Architects* , and *Component Assemblers* . The corresponding process model for a *Component Developer* is rather similar to the process model for component users, as described in Figure 1:

A *Component Developer* does also receive requirements from his customers, although these requirements are specifications which are usually more formal than the requirements provided by end-users. The *Component Developer* screens his stock for a component which suits the customer’s requirements. In some cases an existing component merely has to be adapted during the corresponding design and implementation tasks. The resulting development process is rather fast and the component can soon be delivered.

In other cases, the *Component Developer* has no suitable component in stock, and consequently performs a complete development process as described in the previous section. During *Analysis*, the *Component Developer* should consider customer requirements from a more abstract point of view in order to develop components that may also be (re)used in a different context by different customers. Therefore, a special marketing department should perform an according market study.

During *Business Design* and *Technical Design* the *Component Developer* specifies the component architecture. Possibly, a combination of smaller components within this architecture already fulfills the given requirements. Otherwise, the *Component Developer* has to design and implement the component from scratch. Subsequently, the developed component is tested against the more abstract requirements provided by the market study. Finally, the *Component Developer* will adapt the component, test it against the original requirements provided by the original customer, and deliver it after a successful test.

Conclusion and Further Work

In this paper, we have outlined an overall methodology for componentware consisting of four essential building blocks: a *Formal System Model* , *Description Techniques* , a *Process Model* , and *Tools* . We have presented a modular and adaptable process model suited for componentware based on an overall result resp. task structure and a set of new roles and tasks performing process patterns to fill the result structure.

We think that this work could be part of the CBSE handbook--Chapter 2.1 and 2.2 [CBS99]. We know that we still need to provide further work, especially with respect to refining and elaborating a more detailed result structure. The final result structure will be a combination of the result structures in existing process models, like the RUP [Iva99] or the V-Model [IAB98], tailored to the specific needs of componentware. It will also be elaborated and enhanced with additional aspects, especially with respect to economical and management-related aspects. Based on this result structure, we have to work out the life-cycle and typical activities in componentware projects. Then we have to elaborate a clear understanding about the different roles and responsibilities in component-based development projects.

Finally, the proposed process model and its accompanying pattern language (cf. [BRSV98a,ABD+99]) are far from being complete--both structure and content of the pattern catalog are not sufficiently elaborated. For the CBSE handbook we have to expand and improve the patterns. Furthermore, we have to present existing process models as process patterns, as we already did in [ABD+99]. Thus, the CBSE handbook will contain a set of process patterns which can be tailored individually to the specific needs of the current project.

=

References

ABD+99
Dirk Ansorge, Klaus Bergner, Bernd Deifel, Nicholas Hawlitzky, Andreas Rausch, Marc Sihling, Veronika Thurner, and Sascha Vogel.
Managing componentware development - software reuse and the v-modell process.
In *Proceedings of CAiSE '99*, Lecture Notes in Computer Science. Springer, 1999.

BRSV98a
Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig.
A componentware development methodology based on process patterns.
In *PLOP'98 Proceedings of the 5th Annual Conference on the Pattern Languaes of Programs*. Robert Allerton Park and Conference Center, 1998.

BRSV98b
Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig.
An integrated view on componentware - concepts, description techniques, and development process.
In Roger Lee, editor, *Software Engineering : Proceedings of the IASTED Conference '98*. ACTA Press, Anaheim, 1998.

CBS99
CBSE'99.
`http://www.sei.cmui.edu//cbs//icse99//strawman.html`, 1999.

Cop94
J. O. Coplien.
A development process generative pattern language.
In *PLoP '94 Conference on Pattern Languages of Programming*, 1994.

DW98
Desmond D'Souza and Allan Wills.
Objects, Components, and Frameworks with UML: The Catalysis Approach.
to appear, `http://www.iconcomp.com/catalysis`, 1998.

Hin97
Dietrich Hinz.

Die neue HOAI.
Forum Verlag Herkert GmbH, Merching, 1997.

IAB98
IABG.
Das V-Modell, `http://www.v-modell.iabg.de/`, 1998.

Iva99
Ivar Jacobson and Grady Booch and James Rumbaugh.
The Unified Software Development Process.
Addison Wesley, 1999.

=

About this document ...

Componentware - Methodology and Process

The command line arguments were:
latex2html `position.tex`.

The translation was initiated by Andreas Rausch on 3/22/1999

Andreas Rausch
3/22/1999