An Integrated View On Componentware – Concepts, Description Techniques, and Development 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

Abstract

We outline and clarify the essential concepts of the componentware paradigm. Based on the proposed definitions, we introduce a number of useful description techniques, and sketch a flexible process model for component-based development. The presented techniques and concepts serve as building blocks of an overall methodology for componentware which is the focus of our current work.

Keywords: Componentware, Methodology, Description Techniques, Process Model

1 Introduction

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. Obviously those systems require a more advanced way of structuring, describing and developing them. Componentware is a possible approach to solve these problems.

Although a variety of technical concepts and tools for component-oriented software engineering already exist, 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 [1] as well as textual notations like interface specifications expressed in CORBA IDL [2, 3], C++, or Java [4]. 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 shipping reusable components.

In the following sections, we cover the individual aspects in more detail. First, we provide an overview about important characteristics and concepts of componentware. Based on this, we present description techniques for components and component-based systems, and propose a development process for component users and component developers. Finally, we discuss the necessary requirements for tools supporting and implementing the proposed methodology.

2 Essential Characteristics

The main goals of the componentware approach, namely, building well-structured systems consisting of understandable and reusable parts, are not new in computer science, but have been stressed by earlier paradigms like object-orientation. However, a satisfactory scientific approach to componentware is still missing, quite in contrast to object-orientation, where the basic concepts have settled to some extent during the last years. In the following, we identify,

* This paper originates from the research project *A1 – Component-Based Software Engineering*, a part of *Bayerischer Forschungsverbund Software-Engineering (FORSOFT)*, supported by Siemens ZT.

characterize, and define essential concepts common to a wide range of current componentware approaches.

2.1 Interfaces and Interface Descriptions

The concept of a component interface is central to componentware. It allows to use the functionality of a component only through clearly defined access points.

> A component provides *export interfaces* to offer specific services to other components and it uses *import interfaces* to access services provided by other components. In addition, there may be *combined interfaces* combining the properties of export and import interfaces.

While most current formalisms (e.g. CORBA IDL [2]) only allow to specify the signature of component interfaces, this is not sufficient to reach beyond verification of syntactical correctness of a component system. There is obviously a need for additional information about the semantics and the behavior of interfaces:

> An *interface description* consists of
>
> - a *signature* part, describing the operations provided and used by a component, and, based on that,
> - a *behavior* part, describing the component's dynamic behavior with respect to its interfaces.
>
> An interface description may specify the behavior with respect to a single interface or with respect to a number of interfaces that belong together.

Components may provide one or more standard interfaces for commonly used services like, for instance, configuration, transaction, persistence, or scripting. A very important example is the standard interface that may be queried for the signatures of the component's interfaces. This special functionality is often used by tools to visualize the interfaces of a component during development. The concept of standard interfaces results in two main benefits:

- Separation of concerns reduces the complexity for component developers as they may restrict themselves to clearly defined subsets of the overall functionality.
- The use of well-known standard interfaces helps component users to understand the different capabilities of the entire component more easily.

2.2 Component Types and Instances

Current technical component approaches often do not explicitly distinguish between the concepts of a *component instance* and a *component type*. Formally put, a component type describes the features common to the set of its component instances. In our opinion, the clear separation of the instance and type concepts is necessary to gain a clear understanding and a precise, non-ambiguous conceptual foundation for non-typed component systems as well as for typed ones.

> A *component type* is described by a set of interface descriptions. A *component instance* and its interfaces obey the restrictions of the corresponding type and its interface descriptions. In addition, a component instance has a data state, and the component instance as well as its interfaces can be uniquely identified in the considered context.

Often, there only exists a single component instance of a kind as, for example, in the case of large technical components or legacy systems. In those situations, a component type is most likely not explicitly available at all. Using instance-based visual tools like, for example, Microsoft's Visual Basic [5] or SUN's Java Studio [6], visually represented component instances are placed on a visual form and their interfaces are connected. In contrast, using CORBA IDL [2] requires to specify the interfaces common to all component instances of a single component type. Componentware approaches like Java Beans [7] rely on a package of component instances and their corresponding types.

2.3 Connecting Components

The vision of componentware is also about a new way of programming. The most important feature is the composition of existing, reusable components via their interfaces. At this point, a rather vague definition is sufficient:

> Interfaces of components can be *connected* to each other if their corresponding interface descriptions are compatible.

Of course, the question of compatibility of interface descriptions has still to be solved. As the answer depends on the precise syntax and semantics of the description techniques used to specify the component's signature and behavior, this is a very demanding task requiring a precisely defined mathematical foundation.

As noted above, the connection structure of a component system is usually created during development, when the developer adds new connections between component instances with the help of a visual tool. However, static connection structures between components are not sufficiently powerful to model all aspects of large systems—there is also the need to create and destroy components as well as their connections at runtime. In Section 3 we provide description techniques to model some of the static and dynamic aspects of component connection structures.

2.4 Component Aggregation

The concepts presented so far only allow to model “flat” graphs of connected components.In many cases, this is not sufficient as more structure is needed to manage the involved complexity. This may be achieved by using the concept of aggregation:

> Component instances may *aggregate* other component instances and rely on their functionality to implement their own services.

Usually, multiple containment is not allowed, resulting in a tree as the dominant containment structure. The aggregation structure together with the connection structure is often called the system’s *architecture*. Containment requires additional information about component instances, namely

- which component instance(s) it relies on,
- how the encapsulated component instances are connected to each other, and
- in which way the resulting functionality is achieved.

3 Description Techniques

In the next sections, we will present a family of related description techniques tailored to componentware. They describe

- component types and their corresponding interfaces,
- component instances and the runtime connections between their interfaces,
- graphs of component types,
- navigation between a component’s interfaces, and

We introduce these different description techniques in the context of an exemplary system meant to display text documents. It consists of instances of two component types: Document and View components. Document components encapsulate the content of a text document. They offer functionality to change the content of the text documents and use a global storage mechanism to make the content of the documents persistent. View components display the content of their corresponding document components. They are connected to the document components via an observer mechanism. Later on, this simple example is extended to illustrate a CompoundDocument component framework, in which every document part of the compound document has exactly one main view and an arbitrary number of other, additional views.

3.1 Component Type Diagram

Component Type Diagrams (CTDs) are used to visualize the set of interfaces belonging to a component type. They are an adapted version of UML’s class diagrams (cf. [1]) which represent classes as rectangles and interfaces as circles connected with lines. Figure 1 shows the Document component type of the examplary system and the three interfaces that belong to instances of this component type.

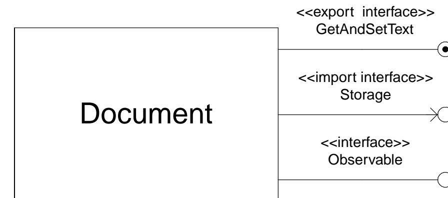

Figure 1: Component Type Diagram

As the document encapsulates the text data it has to export an interface to change the document’s content, namely the GetAndSetText export interface. The document component also has to store the text data persistently. Therefore, it imports the interface Storage. In CTDs, export interfaces are visualized as circles with a dot inside; import interfaces are represented by a circle with an arrow. The document component also provides the combined interface *Observable*, represented by a simple circle with neither dot nor arrow. According to the standard observer mechanism, it exports a method for registration of observers and imports a method for notifying the registered observers (cf. [8]).

The signature of the interfaces may be described by additional UML class diagrams [1]. To specify the behavior part of interfaces, state transition diagrams or sequence diagrams as defined within the UML may be used. A complete description, including the syntactical and behavior part of the combined interface Observable can be found in [9].

3.2 Component Instance Graph

Component Instance Graphs (CIGs) are used to visualize component connection structures arising at runtime. Analogous to UML instance diagrams [1], component instances are shown by using underlined names. Figure 2 shows the connection structure between a single component instance of type Document and a single component instance of type View. The connection itself is visualized by a line connecting the corresponding interfaces.

3.3 Component Type Graph

Component Instance Graphs represent only snapshots of a component system. As long as the connection structure of a system is static and the number of components is not

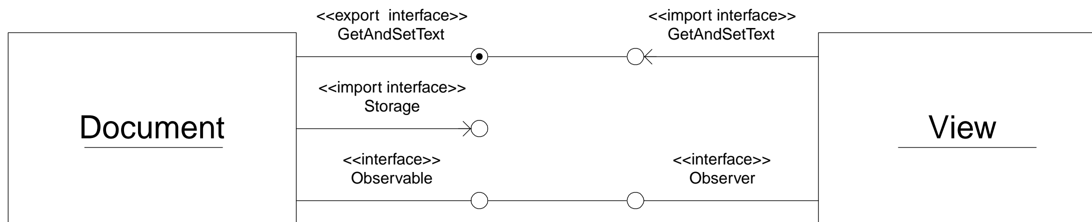

Figure 2: Component Instance Graph

too high, this is an adequate modeling technique. However, it fails when many components are involved or when the connections between the components change dynamically during runtime of the system. In this case, a description technique visualizing the possible connections based on the component types is more adequate. The resulting diagrams are a component-oriented variant of the object-oriented class diagrams.

In our example, we want to model that several views may be connected to a single document component. This is illustrated by the Component Type Graph in Figure 3. As known from class diagrams, it specifies the admitted multiplicities for the dynamic connection structure. An underlying run-time system could verify that this specification is not violated during runtime.

3.4 Interface Navigation Diagram

Interface Navigation Diagrams (INDs) are an adapted and enhanced version of CTDs. INDs specify the mulitplicity of interfaces and the navigation between several interfaces supported by a single component.

For example, if a view component imports the GetAndSetText interface of the document component and later needs to access the Observable interface to register itself as an observer, it may use an operation called getObservableInterface. Figure 4 represents this relationship by a simple arrow between these two interfaces with a label containing the name of the operation. The direction of the arrow shows the navigation direction between the involved interfaces. It is also possible to specify a multiplicity by adding a number to the arrow. This defines how many interface references may be provided to other components.

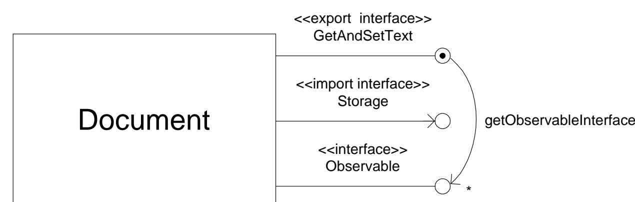

Figure 4: Interface Navigation Diagram

In this paper, we present only a simple version of INDs. The extended version of INDs provides a rich set of concepts, including externally visible classes, their inheritance relations, visible methods, and multiplicities of possible interface instances, determining the maximum cardinality of a set of interfaces at runtime. This version of INDs (called *Component Interface Diagrams* and using a slightly different syntactical representation) as well as a mapping from INDs to practical component approaches, like CORBA [2], COM [10], and Java Beans [7], is described in [11].

4 Process Model

A process model supports system development by clearly defining individual development tasks, roles and results as well as the relationships between them. In the context of componentware, traditional process are not suitable because of the following three issues:

New Roles and Results To leverage the technical advantages of componentware and to support the reuse of existing components, the introduction of new results elaborated by individuals or groups in new roles is immanent. This leads, for example, to the separation of the roles component developer and component assembler, and to new development results like a component repository, a market study, or an evaluation document for interesting commercial components.

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 during a project. 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. With componentware, this issue is very important, as changes on the component market, for example, may require changing the development process.

Combining Top-Down and Bottom-Up Development Traditional top-down approaches start with the initial

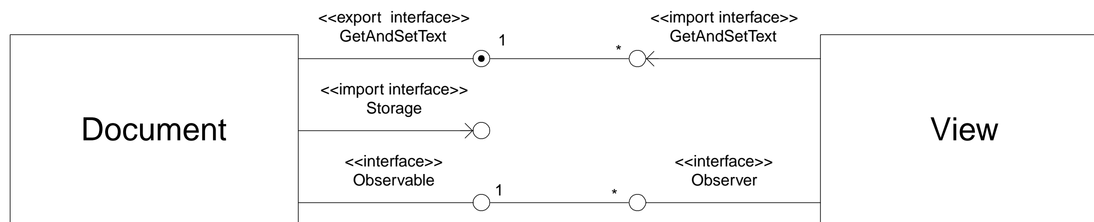

Figure 3: Component Type Graph

customer requirements and refine them continually until the level of detail is sufficient for implementation. Pure top-down development usually leads to systems that are brittle with respect to changing requirements because the system architecture and the involved components are specifically adjusted to the initial set of requirements. This is in sharp contrast to the idea of building a system from truly reusable components.

A bottom-up approach, on the other hand, starts with existing, reusable components. They are iteratively combined and composed to higher-level components until, finally, a top-level component emerges which fulfills the customer's requirements. Obviously pure bottom-up approaches are impractical in most cases because the requirements are not taken into account early enough.

A componentware process model should, therefore, allow for combining top-down and bottom-up aspects as needed in the actual situation of the project.

We propose to base a componentware process model upon the notion of so-called *process patterns* [12], resembling the well-known design patterns [8]. However, a process pattern doesn't suggest a certain solution to a design problem in the context of a system to be developed, but instead proposes a certain process as a reaction to a management problem in the context of the actual development project situation. The actual situation is defined mainly by the current state of the result structure, comprising the description techniques presented above.

An example for a process pattern is the pattern *Component Assessment*. It can be used in project contexts in which the design subresults are finished already. The solved problem is to decide whether a new component on the market should be integrated into the system. The proposed solution comprises various test and evaluation activities of the system architect and the component assembler.

5 Conclusion and Future Work

In this paper, we have outlined an overall methodology for componentware together with its essential building blocks. We are currently refining and elaborating this methodology.

At the current time, parts of the conceptual framework have been defined informally only, and the definition of a complete formal system model for componentware is still work in progress. This also applies to the description techniques—although we performed several case studies in this area, thereby defining some of the mentioned description techniques and their consistency criteria informally [13, 14, 11], a consistent toolkit of conforming, formally founded description techniques is still missing. This work will be firmly based on existing results in the context of the Focus project and the SysLab group [15, 16, 17, 9, 18].

The overall structure of the proposed process model has been described in a report [12]. It will also be elaborated and enhanced with additional aspects, especially with respect to economical and management-related aspects.

A further area of interest is tool support. We participate in the development of a component-oriented architecture for the repository of the CASE tool AutoFocus [19, 20], concentrating on aspects like strong support for versioning, distributed work, and different description techniques [21, 22]. As a long-term goal, we plan to integrate our new description and development techniques into this tool, yielding an integrated platform for componentware development.

Acknowledgements

We thank Bernhard Deifel, Cornel Klein, and Sascha Molterer for interesting discussions and comments on earlier versions of this report.

References

[1] U. Group, "Unified Modeling Language," Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[2] O. M. G. CORBA, "OMG website, http://www.omg.org."

[3] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 2nd ed., 1998.

[4] D. Flanagan, *Java in a Nutshell*. O'Reilly & Associates, Inc., 2nd ed., 1996.

[5] M. Corporation, "MicroSoft VisualBasic homepage, http://www.microsoft.com/vbasic," 1998.

[6] S. Microsystems, "Java Studio homepage, http://www.sun.com/studio," 1998.

[7] JavaSoft, "JavaBeans website, http://java.sun.com/beans/," 1999.

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

[9] M. Broy, C. Hofmann, I. Krüger, and M. Schmidt, "A graphical description technique for communication in software architectures," Tech. Rep. TUM-I9705, Technische Universität München, 1997.

[10] M. Corporation, "MicroSoft COM homepage, http://www.microsoft.com/com," 1998.

[11] F. Huber, A. Rausch, and B. Rumpe, "Modeling dynamic component interfaces," in *Proceedings of TOOLS'98, to appear*, 1998.

[12] K. Bergner, A. Rausch, M. Sihling, and A. 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.

[13] K. Bergner, A. Rausch, and M. Sihling, "Using UML for modeling a distributed Java application," Tech. Rep. TUM-I9735, Technische Universität München, Institut für Informatik, 1997.

[14] A. Vilbig, B. Deifel, S. Molterer, A. Rausch, and M. Sihling, "Using the SysLab method - a case study," Tech. Rep. TUM-I9751, Technische Universität München, 1997.

[15] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber, "The design of distributed systems - an introduction to FOCUS," Tech. Rep. TUM-I9202, Technische Universität München, 1992.

[16] B. Rumpe, C. Klein, and M. Broy, "Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme – das SysLab Systemmodell," Tech. Rep. TUM-I9510, Technische Universität München, Institut für Informatik, 1995.

[17] M. Broy, "Towards a mathematical model of a component and its use," in *Componentware Users Conference 1996, Munich, Proceedings*, SIGS Publications, 1996.

[18] V. Thurner, "A formally founded description technique for business processes," Tech. Rep. TUM-I9753, Technische Universität München, 1997.

[19] F. Huber, B. Schätz, and K. Spies, "Autofocus – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme," in *Formale Beschreibungstechniken für verteilte Systeme* (U. Herzog and H. Hermanns, eds.), pp. 165–174, Universität Erlangen-Nürnberg, 1996.

[20] F. Huber, B. Schätz, A. Schmidt, and K. Spies, "Autofocus - a tool for distributed systems specification," in *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems* (B. Jonsson and J. Parrow, eds.), pp. 467–470, LNCS 1135, Springer Verlag, 1996.

[21] K. Bergner, F. Huber, A. Rausch, and M. Sihling, "Component-oriented redesign of the CASE-tool AutoFocus," Tech. Rep. TUM-I9752, Technische Universität München, 1997.

[22] K. Bergner, F. Huber, A. Rausch, and M. Sihling, "A component-oriented system architecture for the CASE-tool AutoFocus," 1998. Submitted to IASTED SE'98.