

Putting the Parts Together – Concepts, Description Techniques, and Development Process for Componentware*

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>

Abstract

We outline and clarify the essential concepts of the componentware paradigm. After motivating the role of formal foundations and introducing a number of useful description techniques, we propose a flexible process model for component-based development based on process patterns. 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, Pattern

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. Those systems obviously require a more advanced means of structuring, describing and developing them. Componentware is a possible approach to solve to these problems.

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

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. Corresponding informal descriptions are useful to illustrate them. The contained definitions and concepts should be as simple as possible, yet sufficiently powerful to capture the essential concepts and development techniques of existing techni-

cal 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 [17] as well as textual notations like interface specifications expressed in CORBA IDL [11, 29], C++, or Java [14]. 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.

These essential parts of a componentware methodology and their relationships are outlined in Figure 1.

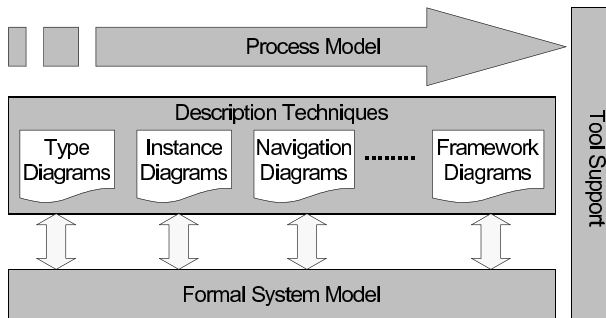


Figure 1. Componentware Methodology

In the following sections, we cover individual aspects in more detail. First, we provide an informal description of important characteristics and concepts of componentware. Based on this, we present an overview of a more rigorous, formal system model for componentware as well as suitable description techniques for components and component-based systems. Finally, we propose a development process for both component users and component developers.

2 Essential Characteristics

The main goals of componentware, 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, 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 [11]) 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 and transaction management, 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 them-

selves 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 [12] or SUN's Java Studio [28], visually represented component instances are placed on a visual form and their interfaces are connected. In contrast, using CORBA IDL [11] requires to specify the interfaces common to all component instances of a single component type. Componentware approaches like Java Beans [25] 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-based 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 4 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 Formal System Model

In the former sections, we informally sketched the essential concepts of componentware. Only if these concepts and their relationships are precisely defined we can expect that

- they are understandable and thus helpful for the software developer,
- we can reason about the behavior of a system built out of components, and
- proper tool support is achievable.

In order to provide a clear characterization of the basic concepts, we have developed a rigorous mathematical system model of component-based information processing systems as a basis for our current work [5]. It represents an adapted version of the system model of the Sys-Lab project [31, 16], covering only the aspects relevant

to componentware. The basic idea is to map components and their interfaces to stream processing functions, whose syntactical and behavioral compatibility may be verified mathematically [8].

Since the mathematical system model provides only a very elementary notion of an information processing system and because it models all relevant aspects at once, it is very difficult to use it directly. For practical development, the use of various textual and graphical description techniques has proven to be valuable [24, 6].

Following Denert [13], we understand software development as a document-oriented activity. A *development document* is a textual or graphical representation describing certain properties of an information system or its parts. This notion encompasses code-oriented documents like sources and binary programs as well as textual documentation, exemplary program execution logs, test cases, interface descriptions, and so on. Of course, it also includes description techniques, like transition diagrams, class diagrams, and sequence diagrams, of modern modeling languages like UML [17, 7].

By providing development documents with different views (“abstractions”) onto a system, each focusing on special characteristics or structures, the inherent complexity of large systems can be reduced to a manageable and understandable size.

Having defined a formal notion of a system, we are able to interpret each document as a predicate over the class of systems formulated in terms of the mathematical system model. This is accomplished by associating an interpretation function with each description technique, mapping well-formed documents (i.e. syntax) to predicates over systems (i.e. semantics). Thereby, we arrive at an integrated semantics for all development documents. This has the following practical advantages:

- We know precisely which proposition about a system is represented by the document,
- we are able to define consistency conditions between documents, and
- we can check whether changes or adaptations of a particular component are allowed in a particular context.

This approach has been successfully applied to several other description techniques (see [23, 18, 32, 30, 10, 26]). Successful mappings from selected description techniques to the proposed system model seem to support our approach [5].

4 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
- connection structures in the context of component frameworks.

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.

4.1 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. [17]) which represent classes as rectangles and interfaces as circles connected with lines. Figure 2 shows the **Document** component type of the exemplary system and the three interfaces that belong to instances of this component type.

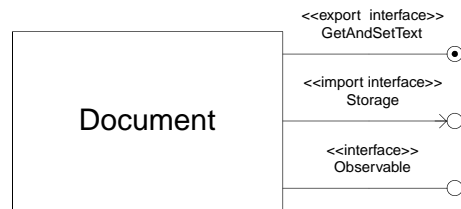


Figure 2. Component Type Diagram

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. [15]).

The signature of the interfaces may be described by additional UML class diagrams [17]. 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 [10].

4.2 Component Instance Graph

Component Instance Graphs (CIGs) are used to visualize component connection structures arising at runtime. Analogous to UML instance diagrams [17], component instances are shown by using underlined names. The connection between interfaces of component instances is visualized by a line, similar to Component Type Graphs as illustrated in the following section.

4.3 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 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.

4.4 Interface Navigation Diagram

Interface Navigation Diagrams (INDs) are an adapted and enhanced version of CTDs. INDs specify the multiplicity 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.

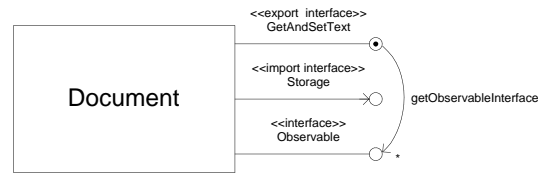


Figure 4. Interface Navigation Diagram

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 [11], COM [27], and Java Beans [25], is described in [20].

4.5 Component Framework Diagram

Recently, the notion of a *component framework* has gained strong attention. According to [33], a component framework “accepts component instances as ‘plug-ins’. A component framework is thus a deliverable on its own that can enforce (sub)system-wide properties of a component system. As such, a component framework is sharply distinct from application frameworks that are subject to (partial) whitebox reuse and that do not retain an identity of their own in deployed systems.”

Informal definitions like this state some of the desired requirements of a component framework—they are not intended as a definition. We propose to define a component framework as a special kind of aggregate component relying on other, aggregated components that are connected according to a certain, specified scheme. The corresponding description technique are *Component Framework Diagrams*, describing the sets of aggregated component instances and their possible connections in the context of the component framework’s type.

Figure 5 shows the Component Framework Diagram of the component framework type `CompoundDocument`. Component frameworks of this type contain a set of arbitrarily many aggregated component instances part of type `Document` as well as two sets `main` and `other` of arbitrarily many instances of type `View`. The multiplicities between these component sets specify that in the context of a `CompoundDocument` every part has to have exactly

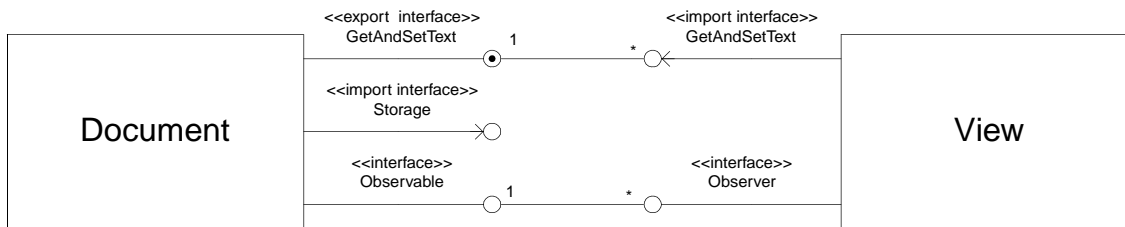


Figure 3. Component Type Graph

one main view and may have arbitrarily many additional views. The **Storage** interface is exposed to the environment of the **CompoundDocument**, to enable persistent storage of the **CompoundDocument** and its **Document** parts. Note that access from other components outside the **CompoundDocument** to the aggregated component instances is not possible in this example because no corresponding export interfaces are exposed.

5 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 this section, we first state 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.

5.1 Essential Aspects of a Component-Oriented Process Model

The characteristics of componentware as described in Section 2 require a suitable process model to address the following areas:

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 leads, for example, to the separation of the roles component developer and component assembler, and to new tasks like searching for existing components and evaluating them well before the final system integration.

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.

Combining Top-Down and Bottom-Up Development

An area where traditional process models have to be improved is their support for simultaneous top-down and bottom-up development:

Traditional top-down approaches start with the initial customer requirements and refine them continually until the level of detail is sufficient for implementation. This process model involves some severe drawbacks: Initially the customer often does not know all relevant requirements, cannot state them adequately, or even states inconsistent requirements. Consequently, many delivered systems do not meet the customer's expectations. In addition, top-down development leads to systems that are very 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.

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.

5.2 Roles of a Component-Oriented Process Model

The distinction between the roles of component developers and component users is a key aspect of a

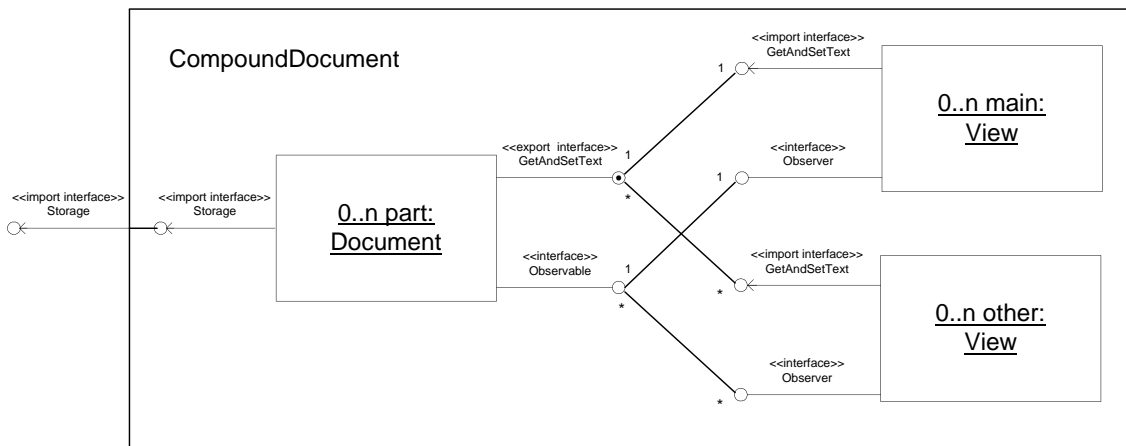


Figure 5. Component Framework Diagram

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 know this separation for a long time [19]. 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.

5.3 Process Model for Componentware

Figure 6 illustrates our proposal for a flexible, component-oriented process model. It applies to *Component Users*, i.e. the developers of component-oriented systems, as well as for *Component Developers* shipping components to the *Component Users*.

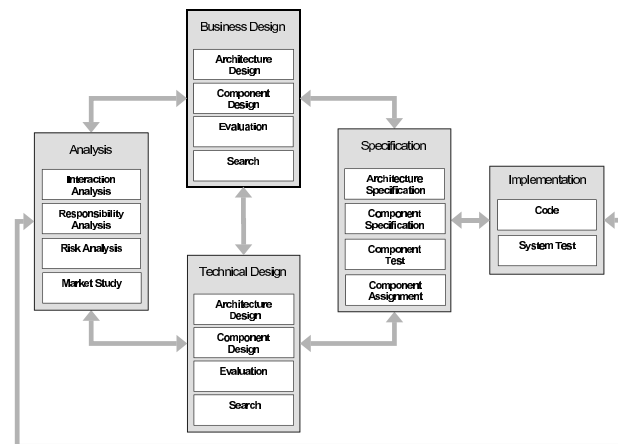


Figure 6. Process Model

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 Archi-

ecture 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 Section 4 and which is reflected in the result structure shown in Figure 6.

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 6.

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 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 external conditions we introduce so-called process patterns which provide guidelines about the next possible development steps. Details about the proposed result structure and the pattern-based approach are described in [4]. In the following, we describe 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**, on the other hand, 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 pre-selection of potentially suiting 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 pre-selection 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.

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.

5.4 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 6:

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.

6 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.

Although a first version of a complete formal system model for componentware has been proposed, the formalization of description techniques is still work in progress. However, we performed several case studies in this area, thereby defining some of the mentioned description techniques and their consistency criteria both formally and informally [3, 35, 20]. The future work will be firmly based on existing results in the context of the Focus project and the SysLab group [9, 31, 8, 10, 34].

The overall structure of the proposed process model has been described in a report [4]. 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 [22, 21], concentrating on aspects like strong support for versioning, distributed work, and different description techniques [1, 2]. 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] K. Bergner, F. Huber, A. Rausch, and M. Sihling. Component-oriented redesign of the CASE-tool AutoFocus. Technical Report TUM-I9752, Technische Universität München, 1997.
- [2] 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.
- [3] K. Bergner, A. Rausch, and M. Sihling. Using UML for modeling a distributed Java application. Technical Report TUM-I9735, Technische Universität München, Institut für Informatik, 1997.
- [4] 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 Languages of Programs*. Robert Allerton Park and Conference Center, 1998.
- [5] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In *Foundations of Component-Based Systems*. Murali Sitaraman and Gary Leavens, to appear.
- [6] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*. Springer Verlag, LNCS 1357, 1997.
- [7] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.
- [8] M. Broy. Towards a mathematical model of a component and its use. In *Componentware Users Conference 1996, Munich, Proceedings*. SIGS Publications, 1996.
- [9] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, 1992.
- [10] M. Broy, C. Hofmann, I. Krüger, and M. Schmidt. A graphical description technique for communication in software architectures. Technical Report TUM-I9705, Technische Universität München, 1997.
- [11] O. M. G. CORBA. OMG website, <http://www.omg.org>.
- [12] M. Corporation. Microsoft VisualBasic homepage, <http://www.microsoft.com/vbasic>, 1998.
- [13] E. Denert. Dokumentenorientierte Software-Entwicklung. *Informatik Spektrum*, 16(3):159–164, June 1993.
- [14] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 2nd edition, 1996.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [16] R. Grosu, C. Klein, and B. Rumpe. Enhancing the SysLab system model with state. Technical Report TUM-I9631, Technische Universität München, Institut für Informatik, 1996.
- [17] U. Group. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.
- [18] R. Hettler. *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen*. PhD thesis, Technische Universität München, Institut für Informatik, 1995.
- [19] D. Hinz. *Die neue HOAI*. Forum Verlag Herkert GmbH, Merching, 1997.
- [20] F. Huber, A. Rausch, and B. Rumpe. Modeling dynamic component interfaces. In *Proceedings of TOOLS'98, to appear*, 1998.
- [21] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus - a tool for distributed systems specification. In B. Jonsson and J. Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.
- [22] F. Huber, B. Schätz, and K. Spies. Autofocus – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. In U. Herzog and H. Hermanns, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996.
- [23] H. Hußmann. Formal foundations for SSADM — an approach integrating the formal and pragmatic worlds of requirements engineering. Technical Report 94-02, FAST (Forschungsinstitut für angewandte Software-Technologie), 1994.
- [24] H. Hußmann. Indirect use of formal methods in software engineering: Position statement. In *International Conference on Software Engineering (ICSE-17), Proceedings Of Workshop on Formal Methods Applications in Software Engineering Practice*, 1995.
- [25] JavaSoft. JavaBeans website, <http://java.sun.com/beans/>, 1999.
- [26] C. Klein. *Anforderungsspezifikation mit Transitionssystemen und Szenarien*. PhD thesis, Technische Universität München, Institut für Informatik, 1997.
- [27] Microsoft Corporation. Microsoft COM homepage, <http://www.microsoft.com/com>, 1998.
- [28] S. Microsystems. Java Studio homepage, <http://www.sun.com/studio>, 1998.
- [29] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 2nd edition, 1998.
- [30] B. Rumpe. *Formale Methodik für den Entwurf verteilter objektorientierter Systeme*. PhD thesis, Technische Universität München, Institut für Informatik, 1996.
- [31] B. Rumpe, C. Klein, and M. Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme – das SysLab Systemmodell. Technical Report TUM-I9510, Technische Universität München, Institut für Informatik, 1995.
- [32] B. Schätz, H. Hußmann, and M. Broy. Graphical development of consistent system specifications. In J. W. Marie-Claude Gaudel, editor, *FME'96: Industrial Benefit and Advances In Formal Methods*, pages 248–267. Springer, 1996. Lecture Notes in Computer Science 1051.
- [33] C. Szyperski. WCOP'98 call for papers <http://www.fit.qut.edu.au/~szypersk/wcop98/cfp.html>, 1998.
- [34] V. Thurner. A formally founded description technique for business processes. Technical Report TUM-I9753, Technische Universität München, 1997.
- [35] A. Vilbig, B. Deifel, S. Molterer, A. Rausch, and M. Sihling. Using the SysLab method - a case study. Technical Report TUM-I9751, Technische Universität München, 1997.